2

REPO AD-A246 277 GE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: 12 Nov 1990 to 01 Jun 1993

4. TITLE AND SUBTITLE
Cray Research, Inc., Cray Ada Compiler, Release 2.0, Cray Y-MP (Host & Target), 901112W1.11117

5. FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-433.0891

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Cray Research, Inc., Cray Ada Compiler, Release 2.0, Cray Y-MP under UNICOS Release 5.0 (Host & Target), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

AVF Control Number: AVF-VSR-433.0891
26 August 1991
90-06-26-TEL

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901112W1.11117
Cray Research, Inc.
Cray Ada Compiler, Release 2.0
Cray Y-MP => Cray Y-MP

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 12 November 1990.

Compiler Name and Version:	Cray Ada Compiler Release 2.0
Host Computer System:	Cray Y-MP under UNICOS Release 5.0
Target Computer System:	Cray Y-MP under UNICOS Release 5.0

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901112W1.11117 is awarded to Cray Research, Inc. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

for

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

John P. Solomond

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

ATTACHMENT D

DECLARATION OF CONFORMANCE

Compiler Implementor: Telesoft, AB.
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB, OH 45433-6503
Ada Compiler Validation Capability (ACVC), Version 1.11

Compiler Name:	Cray Ada Compiler
Compiler Version:	2.0
Host Architecture ISA:	CRAY Y-MP
OS & Version#:	UNICOS Release 5.0
Target Architecture ISA:	CRAY Y-MP
OS & Version#:	UNICOS Release 5.0

Implementor's Declaration

I, the undersigned, representing TELESOFT, declare that TELESOFT has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that Cray Research, Inc. is TeleSoft's licensee of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the licensee's corporate name.

Darcy M. Cohen Date: 12-21-90

TELESOFT
Raymond A. Parra, General Counsel

Licensee's Declaration

I, the undersigned, representing Cray Research, Inc. take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that the Ada language compiler listed, and it's host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Sylvia V. Crain Date: 1/2/90

Cray Research, Inc.
Sylvia Crain
Ada Project Manager

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values — for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler
The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.

Ada Compiler Validation Capability (ACVC)
The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.

Ada Implementation
An Ada compiler with its host computer system and its target computer system.

Ada Joint Program Office (AJPO)
The part of the certification body which provides policy and guidance for the Ada certification system.

Ada Validation Facility (AVF)
The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.

Ada Validation Organization (AVO)
The part of the certification body that provides technical guidance for operations of the Ada certification system.

Compliance of an Ada Implementation
The ability of the implementation to pass an ACVC version.

Computer System
A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity
Fulfillment by a product, process or service of all requirements specified.

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values — for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 12 October 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026B
B85001L	C83026A	C83041A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 229 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113J..Y (16 tests)   C35705J..Y (16 tests)
C35706J..Y (16 tests)   C35707J..Y (16 tests)
C35708J..Y (16 tests)   C35802J..Z (17 tests)
C45241J..Y (16 tests)   C45321J..Y (16 tests)
C45421J..Y (16 tests)   C45521J..Z (17 tests)
C45524J..Z (17 tests)   C45621J..Z (17 tests)
C45641J..Y (16 tests)   C46012J..Z (17 tests)

The following 21 tests check for the predefined type SHORT_INTEGER:

C35404B   B36105C   C45231B   C45304B   C45411B
C45412B   C45502B   C45503B   C45504B   C45504E
C45611B   C45613B   C45614B   C45631B   C45632B
B52004E   C55B07B   B55B09D   B86001V   C86006D
CD7101E

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C   C45231C   C45304C   C45411C   C45412C
C45502C   C45503C   C45504C   C45504F   C45611C
C45612C   C45613C   C45614C   C45631C   C45632C
B52004D   C55B07A   B55B09C   B86001W   C86006C
CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35508I..J and C35508M..N (4 tests) include enumeration representation clauses for boolean types in which the specified values are other than (FALSE => 0, TRUE => 1); this implementation does not support a change in representation for boolean types. (See section 2.3.)

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35702B, C35713C, B86001U, and C86006G check for the predefined type LONG_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B, C46034B and CD2A53A contain 'SMALL representation clauses which are not powers of two or ten.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

LA3004A, EA3004C, and CA3004E check for pragma INLINE for procedures.

LA3004B, EA3004D, and CA3004F check for pragma INLINE for functions.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD1009E and CD1009F contain length clauses that effectively require array components to cross storage boundaries; this representation is not supported by the implementation. (See Section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method:

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO

CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E CE2107G..H CE2107L CD2110B CE2110D
CE2111D CE2111H CE3111B CE3111D..E CE3114B
CE3115A

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 30 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests:

BA1001A BA2001C BA2001E BA3006A BA3006B
BA3007B BA3008A BA3008B BA3013A

C35508I..J and C35508M..N (4 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests attempt to change the representation of a boolean type. The AVO ruled that, in consideration of the particular nature of boolean types and the operations that are defined for the type and for arrays of the type, a change of representation need not be supported; the ARG will address this issue in Commentary AI-00564.

C52008B was graded passed by the Test Modification as directed by the AVO. This test uses a record type with discriminants with defaults; this test also has array components whose length depends on the values of some discriminants of type INTEGER. On compilation of the type declaration, this implementation raises NUMERIC_ERROR as it attempts to calculate the maximum possible size for objects of the type. Although this behavior is accepted for validation in consideration of intended changes to the standard to allow for compile-time detection of run-time error conditions. The test was modified to constrain the subtype of the discriminants. Line 16 was modified to declare a constrained subtype of INTEGER, and discriminant declarations in lines 17 and 25 were modified to use that subtype; the lines are given below:

```
16   SUBTYPE SUBINT IS INTEGER RANGE -128..127;
17   TYPE REC1(D1,D2 : SUBINT) IS

25   TYPE REC2(D1,D2,D3,D4 : SUBINT := 0) IS
```

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies contain uses of the types that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete—no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended errors were then detected by the compiler.

CD1009A, CD1009I, CD1C03A, CD2A21A, CD2A22J, CD2A23A, CD2A24A, and CD2A31A..C (3 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests use instantiations of the support procedure LENGTH_CHECK, which uses Unchecked_Conversion according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instances of LENGTH_CHECK—i.e., the allowed Report. Failed messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

CD1009E and CD1009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests use length clauses for array types with type INTEGER components that specify the size to be 'LENGTH * INTEGER'SIZE; for this implementation INTEGER'SIZE is 46 (bits) and SYSTEM.STORAGE_UNIT is 64 (bits)—one machine word—; hence, the specified representation cannot be met without some of the array components crossing word boundaries, which this implementation does not support.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Sylvia Crane
Cray Research, Inc
500 Montezuma, Suite 118
Santa Fe NM 87501

For a point of contact for sales information about this Ada implementation system, see:

Sylvia Crane
Cray Research, Inc
500 Montezuma, Suite 118
Santa Fe NM 87501

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3715
b) Total Number of Withdrawn Tests	81
c) Processed Inapplicable Tests	145
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	229
f) Total Number of Inapplicable Tests	374
g) Total Number of Tests for ACVC 1.11	4170

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 373 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 229 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

```
-O 2

-S

-W +vector=OFF

-W +enable_traceback

-m
```

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Sylvia Crane
Cray Research, Inc
500 Montezuma, Suite 118
Santa Fe NM 87501

For a point of contact for sales information about this Ada implementation system, see:

Sylvia Crane
Cray Research, Inc
500 Montezuma, Suite 118
Santa Fe NM 87501

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string_aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	200
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values:

Macro Parameter	Macro Value
$ACC_SIZE	22
$ALIGNMENT	1
$COUNT_LAST	3518437208829
$DEFAULT_MEM_SIZE	16#7FFFFFFE#
$DEFAULT_STOR_UNIT	64
$DEFAULT_SYS_NAME	CRAY_YMP
$DELTA_DOC	2#1.0#E-45
$ENTRY_ADDRESS	16#40#
$ENTRY_ADDRESS1	16#80#
$ENTRY_ADDRESS2	16#100#
$FIELD_LAST	1000
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.80141E+38

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	63
$ILLEGAL_EXTERNAL_FILE_NAME1	"BADCHAR*^/%"
$ILLEGAL_EXTERNAL_FILE_NAME2	"/NONAME/DIRECTORY"
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	'PRAGMA INCLUDE ("A28006D1.TST");'
$INCLUDE_PRAGMA2	'PRAGMA INCLUDE ("B28006F1.TST");'
$INTEGER_FIRST	-3518437208832
$INTEGER_LAST	3518437208831
$INTEGER_LAST_PLUS_1	3518437208832
$INTERFACE_LANGUAGE	C
$LESS_THAN_DURATION	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST	-131_073.0
$LINE_TERMINATOR	ASCII.LF
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	45
$MAX_DIGITS	13
$MAX_INT	3518437208831
$MAX_INT_PLUS_1	3518437208832
$MIN_INT	-3518437208832
$NAME	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST	CRAY_YMP

$NAME_SPECIFICATION1	/tmp/X2120A
$NAME_SPECIFICATION2	/tmp/X2120B
$NAME_SPECIFICATION3	/tmp/X3119A
$NEG_BASED_INT	16#3FFFFFFFFFFE#
$NEW_MEM_SIZE	16#FFFFFFFF#
$NEW_STOR_UNIT	64
$NEW_SYS_NAME	CRAY_YMP
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	64
$TASK_STORAGE_SIZE	4096
$TICK	10.0E-3
$VARIABLE_ADDRESS	16#0020#
$VARIABLE_ADDRESS1	16#0024#
$VARIABLE_ADDRESS2	16#0028#
$YOUR_PRAGMA	PRESERVE_LAYOUT

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to linker documentation and not to this report.

You may also want to compile a series of source files by using an input list file. See "Input file: *input_spec*," page 98, for more information.

The ada command
3.3

After you have created a working library, as described in "Creating sublibraries: the acr command," page 33, you are ready to use the Cray Ada compiler.

The ada command invokes the compiler. Unless you specify otherwise, the front end, middle pass, and code generator are executed each time the compiler is invoked. Specifying the -e option invokes only the front end.

The general command format for invoking the Ada compiler when the -e argument is used is as follows:

```
     ┌ -l libname            ┐
ada  │                       │ [-e] [-E value] [-v] [-L] [-T opt [ ,opt...]] [-k]
     └ -t sublib[,sublib...] ┘

        [-a  file] input_spec
```

The general command format for invoking the Ada compiler when the -e argument is not used is as follows:

```
     ┌ -l libname            ┐
ada  │                       │ [-E value] [-v] [-L] [-T opt [ ,opt...]] [-a file]
     └ -t sublib[,sublib...] ┘

        [-x] [-k] [-d] [-i key] [-j] [-K] [-u key... ] [-O key [ ,key...] [aopt_opts]]

        [-s] [-f size] [-m unit[ald_opts]] [-S key] input_spec
```

ada command options
3.4

The following table shows the ada command-line arguments grouped according to function.

Table 13. ada command arguments

Argument	Action	Page
Library arguments:		
-l *libname*	Uses *libname* as the library file. This option is not used with -t. Default is liblst.alb.	82
-t *sublib* [,*sublib...*]	Specifies temporary list of sublibraries. This option is not used with -l. Default is no list.	82
Execution control arguments:		
-E *value*	Aborts compilation after *value* number of errors. Default is 999.	82
-m *unit* [*ald_opts*]	Specifies the main program unit for compilation. Without this option, no executable code is generated. You can specify arguments from the ald command in conjunction with this argument.	83
-v	Displays progress messages during compilation. Default is no display of progress messages.	84
Output control arguments:		
-e	Runs only the front-end compiler pass. Default runs front-end pass, middle pass, and code generator.	85
-d	Generates source-level debugging information. Default does not include debug information.	85
-f *size*	Specifies alternate frame package size. Default is 18 words. (CRAY-2 systems only.)	86
-i *key*	Suppresses checks and source information; default is to store source line information in the object code.	86
-k	Retains low form and high form of secondary units; default is to discard them.	87
-O *key* [,*key*] [*aopt_opts*]	Optimizes object code. You can specify arguments from the aopt command in conjunction with this argument. See the -O argument documentation for information on *keys*.	88

Table 13. ada command arguments
(continued)

Argument	Action	Page
-s	Controls sharing of frame packages. By default, there is 1 frame package for each routine and noncalling routines share a package. (CRAY-2 systems only.)	91
-x	Puts Profiler information into generated code; default does not generate execution-profile code.	91
Listing control arguments:		
-a *file*	Specifies alternate list files; by default writes to *file*.l.	91
-j	Joins errors with source code.	92
-K	Keeps source file in library.	92
-L	Generates interspersed source-error listing. Default does not.	92
-S *key*	Generates source/assembly listing. Default does not.	93
-T *opt*[,*opt*...]	Controls terminal display. By default it does not include vectorizer messages; it prints warning-level messages, and it puts one line of code around error messages.	95
-u *key*	Updates the working sublibrary. Default updates the library after all units within a single source file compile successfully.	98
Required argument:		
input_spec	One or more Ada source files or an input list file to be compiled.	98

The following subsections describe the options according to the preceding functional groups.

Command syntax 4.1.1

The valid syntaxes of the ald command are as follows:

```
ald [-l libname] [-b] [-o file] [-d] [-v] [-X] [-x] [-S key]

[-P 'string'] [-p object[,object,...]] [-Y size] main_unit
```

```
ald [-t sublib[,sublib...]] [-b] [-o file] [-d] [-v] [-X] [-x] [-S key]

[-P 'string'] [-p object[,object,...]] [-Y size] main_unit
```

Table 14. ald command options

Option	Action	Default
Library options:		
-l *libname*	Specifies name of library file. Not used with -t.	liblst.alb
-t *sublib*[,...]	Specifies temporary list of sublibraries. Not used with -l.	None
Command options:		
-b	Makes loader quit after creating elaboration code and linking order.	Creates executable module
-o *file*	Specifies name of executable file.	Name of main unit
-d	Includes debugging information; specify if you intend to use the debugger.	None
-v	Makes linker send messages to stdout for each phase of linking.	No messages
-X	Reports unhandled exceptions in tasks.	No reporting of unhandled exceptions
-x	Instructs binder to link in the profiler run-time support routines.	No profiler support linked in

Table 14. ald command options
(continued)

Option	Action	Default
-P 'string'	Passes a string of SEGLDR options to SEGLDR.	No string passed to SEGLDR
-p *obj*[,*obj*...]	Passes name of foreign *object* files to SEGLDR to link with the Ada unit.	Nothing passed to SEGLDR
-Y *size*	Specifies amount of space allocated for each task.	4000 words
-v	Displays progress messages during compilation.	No progress messages displayed
-S *key*	Generates source/assembly listing	No source/assembly listing generated
Required option:		
main_unit	Name of the main Ada program unit to be linked. The *main_unit* name must match the name of the main unit specified in the Ada library. If the *main_unit* name is longer than 14 characters, ald truncates the name to 14 characters. If another file exists with a name equal to the 14-character *main_unit* name, the other file is overwritten because of the 14-character UNICOS 5.1 naming restrictions. In UNICOS 6.0 , the parameters are 255 characters.	None

Command options 4.1.2

The following subsections describe the ald command options.

Creating only elaboration code: -b 4.1.2.1

The -b option causes the Ada loader to terminate after it has created the elaboration code and linking order, before it invokes SEGLDR. This provides more control over the linking process by letting the user modify the link order and add SEGLDR directives. The ald command generates two files; the first file,

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...
   type INTEGER is range -2**45 .. 2**45-1;

   type FLOAT is digits 13 range -1.0E2466 .. 1.0E2466;

   type DURATION is delta 2#1.0#E-14 range -86400.0 .. 86400.0;
   ...

end STANDARD;
```

Portability Considerations [1]

Page

Portability Considerations [1]

Several issues arise with regard to the generation and movement of executable files between various Cray Research systems. Generally, the rules for Cray Ada are as follows:

- Code can be generated only for the specific model on which the Ada compiler is running.
- Ada is compatible only at the level of source code.

The specific issues related to Ada support on the various Cray Research architectures are explained in the following subsections.

CPU targeting 1.1

Cray Ada does not support a mechanism that targets code for a Cray Research system other than the one on which the compilation occurs. For example, there is no user option that lets Ada code compiled on a CRAY-2 system produce executable files for a CRAY Y-MP system.

EMA and no EMA 1.2

The issue of extended memory addressing (EMA) is relevant for CRAY X-MP systems. Currently, the Ada compiler can generate executable code only for the specific system configuration on which the compiler is running.

Users must be extremely careful when attempting to execute code on a machine other than the one on which it was compiled, because the compiler does not currently notify users when the addressing scheme used in compiling a given code is incompatible with or different from that of the target execution machine. To avoid unexpected execution problems or inaccurate results, the best (and recommended) practice is to execute code on only the machine on which it was compiled.

CRAY X-MP system compatibility mode
1.3

Currently, the Cray Ada compiler does not support the TARGET environment variable that indicates system hardware. Users must ensure that the correct libraries are included when linking. "The Ada Linker," in *Cray Ada Environment, Volume 1: Reference Manual*, publication SR-3014, describes several appropriate strategies for referencing alternative libraries. Users should also check with their system support staff to determine whether a transparent user interface has been established for their system.

The most common message that indicates a library compatibility problem resembles the following:

```
*Warning* File '/usr/lib/libc.a' contains 26
modules with conflicting machine
characteristics
```

Runtime support
1.4

The Cray Ada runtime sublibrary, /usr/lib/runtime.sub, has been tailored to support the specific architecture on which it runs. For CRAY X-MP systems, variations in the runtime library relate primarily to issues of EMA and compress-index/gather-scatter hardware support. For CRAY-2 systems, the runtime library differs from that of CRAY X-MP and CRAY Y-MP systems primarily because of differences in the two architectures' instruction sets. Avoid moving the runtime library from one machine to another.

Interfacing to other languages
1.5

When mixing Ada with another language such as Fortran, C, Pascal, or CAL, users must ensure that the hardware attributes (CPU type, 24-bit addressing, and so on) used to compile the foreign modules agree with those of the Ada compilation environment. Compilation of all foreign language modules should be performed on the machine on which the Ada modules were compiled and should use the targeting defaults available with the foreign language compiler.

Limitations on SEGLDR map files
1.6

A limitation exists regarding SEGLDR load maps for very large Ada programs. Specifically, version 5.2 of SEGLDR (`ld`) cannot produce map files for Ada programs containing more than about 16,000 subprograms. By default, `ald` does not request a map file from SEGLDR, so this does not usually cause a problem. However, if you have a very large Ada program and you request a load map (perhaps for use with `cdbx`), either with the `-p` switch to `ald` or by modifying the link script, you should be aware of this restriction.

This limitation was removed in SEGLDR version 6.0 for UNICOS 6.0.

Getting the Most from the Cray Ada Environment [2]

Table

Getting the Most from the Cray Ada Environment [2]

This section provides a number of techniques that you are encouraged to use to get maximum performance and functionality from the Cray Ada Environment.

Compiling specifications and bodies separately 2.1

You should take full advantage of Ada's separate compilation capabilities. In particular, each compilation unit should be located in a separate source file. This ensures that the minimum amount of recompilation of dependent units is required when any particular compilation unit has been modified.

Maintaining the specification and body of a given compilation unit in one source file may appear to be a reasonable alternative to completely separate files. However, if you do this, any recompilation of the body causes the recompilation of the specification, which in turn requires the recompilation of all units that either directly or indirectly import that specification. When such a recompilation involves a low-level package that is widely used, virtually the entire application may have to be recompiled. Similar principles apply to the maintenance of multiple unit specifications in a single source file. In some instances, a valid compilation order cannot be found when multiple units reside in one file.

Using generics efficiently 2.2

Although the Cray Ada compiler does not currently implement code sharing for instantiations, there are several ways to use generic program units effectively. They are discussed in the subsections that follow.

Use preinstantiated generic units 2.2.1

Compiling the instantiation of a generic unit is roughly equivalent to compiling the source of the generic unit itself at the point of instantiation. When the generic unit contains a large volume of code, this operation can use significant machine resources. Moreover, this overhead is incurred each time the instantiating package is compiled.

To minimize overhead, you should preinstantiate generic units by compiling them in the library whenever possible. This is especially true for the standard predefined I/O generics, which are relatively large. For your convenience, `/usr/lib/runtime.sub` contains two generic instantiations from package `Text_IO`. These are `Integer_Text_IO` and `Float_Text_IO`. Additionally, both float and integer instantiations have been provided for `CRAY_LIB`. These are `CRAY_MATH_LIB`, `CRAY_BIT_LIB`, and `CRAY_UTIL_LIB`. See "Library Interfaces," page 119, for additional information on `CRAY_LIB`.

The following example shows the specifications of these packages:

```
with Text_IO;
with CRAY_LIB;

-- Text_IO instantiations
Package Integer_Text_IO is new Text_IO.Integer_IO(Integer);
Package Float_Text_IO is new Text_IO.Float_IO(IFloat);

-- CRAY_LIB Instantiations
Package CRAY_MATH_LIB is new CRAY_LIB.MATH_LIB(Float);
Package CRAY_MATH_LIB is new CRAY_LIB.MATH_LIB(Integer);
Package CRAY_BIT_LIB is new CRAY_LIB.BIT_LIB(Float);
Package CRAY_BIT_LIB is new CRAY_LIB.BIT_LIB(Integer);
Package CRAY_UTIL_LIB is new CRAY_LIB.UTIL_LIB(Float);
Package CRAY_UTIL_LIB is new CRAY_LIB.UTIL_LIB(Integer);
```

You may want to add additional commonly used I/O package instantiations to your own sublibraries and use the preinstantiated versions in your application, besides any of your own generic units. The following is an example of a package containing generic units:

```
with Text_IO;
package Numeric_IO_Stuff is

  package Float_IO is new Text_IO.Float_IO(Float);

  package Int_IO is new Text_IO.Integer_IO(Integer);

end Numeric_IO_Stuff;
```

Instead of reinstantiating new versions of `Integer_IO`, other packages would simply import this package by using the `with Numeric_IO_Stuff` line. This method can save compilation time and code space.

Compile generic bodies early 2.2.2

Although the compiler lets you compile an instantiation of a generic unit at any time after the generic specification has been compiled, it is best not to compile any instantiations until after the corresponding generic body has also been compiled. Otherwise, you will be forced to recompile all instantiations of that generic unit.

Keep generic bodies small 2.2.3

Because the body code is effectively recompiled with each new instantiation of a generic unit, it is best to keep the body code as small as possible. Try to partition the functionality of a generic unit into parts that are specific to the actual parameters of the unit and parts that can be common to different instantiations. The latter can be placed in nongeneric program units that are referenced and called from the generic body. For more information, see the *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

Use pragma INLINE *for optimizing* 2.2.4

Many generics involve subprogram parameters to provide relational operations on generic formal private types. Often, the subprograms supplied as generic actual parameters are trivial. In such cases, they should be marked for inlining during optimization by use of pragma INLINE. Because generic instantiations are macro expansions of the generic body template, calls to generic subprogram parameters can be inlined at the time of the instantiation, avoiding any call overhead.

As specified by the Ada LRM, the pragma must be placed in the same declarative region as the declaration of the subprogram to be inlined and must follow the subprogram declaration. If you do not specify `-O on` or `-O I` as an option on the `ada` or `aopt` command line, pragma INLINE is ignored.

Optimizing hints 2.3

The information in the following subsections will help you achieve better optimization of your code.

Weighted sections of code 2.3.1

The optimizer, when it assigns weights to sections of code, gives a higher weight to the IF part of a branch than to the ELSE part. If one alternative is more likely than the other, arrange the IF statement to take this into account.

Nonnegative variables 2.3.2

Several optimizations are possible when the compiler knows that the range of a variable is strictly nonnegative. Therefore, when possible, declare variables of type Positive or of type Natural. This is particularly important for integer Divide, Rem, and Mod.

Short-circuit Boolean expressions 2.3.3

Short-circuit Boolean expressions that are evaluated into a Boolean variable almost always generate a subroutine call. Therefore, it is best not to use the short-circuit forms in those cases. Short-circuit Boolean expression in `if` statements do not have this problem.

Using pragma ELABORATE 2.4

It is syntactically possible but semantically incorrect to call an Ada subprogram before its body is elaborated. The Ada LRM requires that the PROGRAM_ERROR exception be raised if such a call is made. As a default, the code generated by the compiler must contain checks to ensure that called routines have already been elaborated.

Pragma ELABORATE lets you specify explicit elaboration order dependencies beyond those consistent with the partial ordering defined by with clauses. Pragma ELABORATE must immediately follow the context clause of a compilation unit, before any subsequent library unit or secondary unit.

Pragma ELABORATE specifies that the body of a given library unit must be elaborated before a given compilation unit. For example, using the pocket_calculator example shown in *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014, suppose that a subunit, named sqrt referenced the specification of package trig. In this case, the directive, pragma ELABORATE(trig, calc.arith), inserted after the context clause in sqrt would ensure that the specification of trig would be elaborated before the body of sqrt's parent unit, calc.arith.

The elaboration of an executing program observes these explicit directives in addition to the implicit ordering required by the compilation unit interdependencies. Calls to routines in a compilation unit that is mentioned in pragma ELABORATE do not have to be checked, because the pragma provides the called routine that has already been elaborated. Therefore, the compiler omits elaboration checks and produces smaller, faster code for interunit calls.

Organizing sublibraries 2.5

The organization of sublibraries is an important part of managing an Ada system. The following subsections discuss guidelines for the management of a sublibrary hierarchy.

Restrict the number of sublibraries 2.5.1

Compilation proceeds most rapidly when the number of sublibraries in a library is kept to a minimum. The number of sublibraries has a minimal effect on compilation speed when the number is small, but it can become a significant factor when the number becomes large (more than five).

Restrict the number of units in the library 2.5.2

A compilation rate dependence similar to that for sublibraries but smaller, applies to the number of units in a library. Therefore, if a library contains a large number of small units, it may be advantageous to decrease the total number of units contained in the library.

Therefore, when compiling moderately large systems (100 – 400 compilation units), you should use the following general methodology to enhance compilation rates:

1. Compile all of the unit specifications for the system into one sublibrary.
2. Divide the corresponding unit bodies into two or more groups. This division should be made along functional lines, although an arbitrary division also works.
3. Construct a library file that contains only three sublibraries: an empty sublibrary as the working sublibrary, the sublibrary containing the unit specifications, and the standard predefined runtime sublibrary.
4. Compile a group of unit bodies into this library.
5. Set aside the working sublibrary containing the compiled unit bodies.
6. Repeat steps 3 through 5 until all groups of unit bodies have been compiled.

At this point, you will have one sublibrary containing the compiled specifications and a series of sublibraries containing all of the compiled unit bodies. A library specifying this set of sublibraries may then be used to bind the main program unit into an executable Ada program.

This method enhances the compilation rate because at any time the library contains at most three sublibraries, and the working sublibrary contains only a subset of the compiled bodies. The unit specifications, which determine the Ada compilation order dependencies, are contained in one sublibrary so that the bodies may be compiled in any order.

This method also provides some procedural advantages. Large systems are typically compiled with some form of compilation command file. However, a failure in the compilation of one specification (for example, because of a syntax error) causes other subsequent units that depend on that unit to fail. Thus, either complex logic in the command file or close monitoring by the developer is required to avoid a chain of compilation failures subsequent to the first compilation failure of a unit specification.

With the preceding method, the need for prompt error detection exists only for the compilation of the unit specifications. When the sublibrary containing the unit specifications has been compiled without error, the subsequent body compilations may be allowed to proceed with little or no monitoring; any bodies that fail to compile may have their source code corrected and be recompiled without affecting any of the other compiled units.

Note

Bodies of compilation units that contain many subunits should be treated as specifications, that is, monitored for prompt error detection.

The sublibrary containing the body of a generic unit or the body of a subprogram designated with pragma `INLINE` must be included in the library list when callers or instantiators of the subprogram are compiled.

The preceding scheme may be unfeasible for developing very large systems or, if feasible, not optimal because of the large number of specifications that would have to be held in one sublibrary. An optimal strategy also depends on the stability of the interface specifications.

For very large systems, a four-level hierarchy might be useful; it should include a separate sublibrary for each of the following:

- Predefined runtime library units
- Stable interface specifications common to major functional components
- Specifications local to major functional components
- Bodies for a subgrouping of major functional components

Use multiple sublibraries for unit testing 2.5.3

During program development, you may want to try a modification to the body of an existing compilation unit while saving the previous version of the unit as a backup. For example, you may want to experiment with a new algorithm or insert debugging code. You can accomplish this easily by adding an empty working sublibrary and compiling the new version of the unit body into that sublibrary. As long as the test sublibrary appears in the library file before (above) the sublibrary containing the previous version, the new unit body will effectively replace the old one in subsequent binding operations; the main program must be in the working sublibrary during binding. Therefore, the main program may have to be recompiled or copied into the working sublibrary by use of the acp command.

To return to the previous version, simply remove the test sublibrary from the library list, or move it lower in the list than the sublibrary containing the original version. To replace the previous version with the new version, use the amv command to move the new version from the testing sublibrary into the sublibrary containing the previous version and remove the testing sublibrary from the library file.

Compressing sublibraries
2.6

The Cray Ada Environment compression utility recovers disk space that is lost to internal fragmentation over the course of many recompilations into a sublibrary. Additionally, it orders the database elements to promote efficient I/O operations on the database. Sublibrary compression minimizes library space, permits faster sublibrary access, and improves compilation speed. This feature is particularly important to users having limited disk space, because compressions of more than 50% can be realized. For more information about this feature and the command that invokes it, see *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

Using input-list files to compile
2.7

As mentioned in "Compiling a list of files," in *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014, when you use an input list or name several files to be compiled on the command line, the files in the list are compiled in one compiler invocation. This can provide a savings in compilation time, because the Ada library is opened only once (a relatively costly operation). In some instances, you may achieve compilation-time savings of 30% to 50%.

However, when input-list files contain too many entries, they can slow down the compilation process. This occurs when the compilation becomes too large to compete successfully for the memory space. This threshold depends on the memory size of your Cray Research system and any per-user memory limits set by the system administrator, the system load, the number of entries in the input-list file, and the size of each compilation unit.

Passing parameters efficiently
2.8

For the most efficiency in passing parameters with a CRAY Y-MP or CRAY X-MP system, the most important (most used) parameters should be declared first. For the best performance, users should copy variables defined outside the current procedure into locally defined variables before heavy use situations such as loops.

Taking the 'address of a variable or parameter restricts it from being allocated to a register. On CRAY-2 systems, if any of the parameters are used by variables defined outside the current procedure into locally defined variables or have their 'address taken, this causes all parameters to be written to the stack. This can cause a substantial performance degradation.

System limitations 2.9

The Cray Ada compiler was designed to be largely insensitive to the size of compilation units, in terms of its memory requirements. Under some circumstances, however, the compiler takes an exception of STORAGE_ERROR, indicating insufficient memory to compile a specific compilation unit. In most such cases, breaking the compilation unit into smaller pieces of code should alleviate the problem. For other possible remedies, see *Cray Ada Environment, Volume 1: Reference Manual,* publication SR-3014.

Table 1 shows the Cray Ada compiler's tested capacities. These capacities are currently tested values for a CRAY X-MP/416 system; actual limits may be higher or lower, depending on your hardware configuration.

Table 1. Tested compiler capacities

Category	Item	Capacity value
Ada source code	Length of identifier	200 (or max line width if >80)
	Length of labels	200 (or max line width)
	Length of qualified identifiers	200 (or max line width)
	Length of attributes	200 (or max line width)
	Length of strings	4,194,304
	Maximum dimensionality of arrays	12
	Number of characters per logical line	200
	Number of compilation units	800
	Number of distinct declarations	50,000
	Number of elaboration pragmas	600
	Number of library units	5000
	Number of package names in a `with` clause	16
	Number of packages included (using `with`) in a compilation unit	600
	Number of priorities	64
	Number of simultaneously active tasks	1000
	Depth of nesting of tasks	128
Compilation unit limits	Depth of nesting of blocks	80
	Depth of nesting of `case` statements	128
	Depth of nesting of `if` statements	128
	Depth of nesting of `loop` statements	100
	Depth of nesting of subprograms	24 (CRAY Y-MP, CRAY X-MP) 128 (CRAY-2)
	Number of 64-bit elements in an array	4,194,304 (CRAY X-MP) 2,147,483,648 (CRAY Y-MP, CRAY-2)§
	Number of attributes	100
	Number of declarations	5000
	Number of declarations in a block	1000
	Number of enumeration values in a type	255
	Number of `elsif` alternatives	256
	Number of explicit exceptions	256
	Number of identifiers	10000
	Number of literals	500
	Number of record components	1000
	Number of statements and declarations per procedure	32,767
	Number of subtypes of a type	1099 100 (CRAY-2)

§ This value is tested for compilation only. Ability to use this size is dependent on your systems memory.

Table 1. Tested compiler capacities (continued)

Category	Item	Capacity value
	Number of type declarations	2000
	Number of variant parts	500
Expression limits	Depth of parenthesis nesting	25
	Number of functions	100
	Number of objects	250
	Number of operations	250
Subprogram limits	Number of declarations in a subprogram	1000
	Number of formal parameters	No limit (CRAY Y-MP, CRAY X-MP) 64 (CRAY-2)
Package limits	Number of private declarations/package	1000
	Number of visible declarations/package	1000
Task limits	Number of accepts	50
	Number of delays	25
	Number of entries	25
	Number of select alternatives	25

Interfacing to other languages 2.10

To debug foreign language modules at the source code level you must use cdbx. While you do this, however, the Ada program can be debugged only at the machine level.

If you use adbg to debug an Ada program at the source-code level, debugging of foreign language modules with adbg is available only at the machine level.

Arrays
2.11

The following is a list of guidelines for the allocation of arrays:

- If the array is defined in a procedure, it is dynamically allocated on the stack at runtime.
- If the array is defined by an allocator, it is dynamically defined on the heap at run time.
- If the array is defined in a package which is then `with` ed, it is statically allocated at compile time and is part of the total program field length.

If allocation cannot be done at runtime, a `Storage_Error` will be raised.

Optimization and Vectorization [3]

Tables

Optimization and Vectorization [3]

The optimization features include both significant enhancements to general optimizations and the introduction of general optimizations such as automatic vectorization and scheduling of instructions. You can control these features by using the `-O` *key* option of the `ada` and `aopt` commands.

Several optimization options are available through `aopt` and through pragmas included in your source code. This section describes the use of pragmas. For more information on the `ada` and `aopt` commands, see *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

The types of optimizations supported by Cray Ada are as follows: general optimizations, instruction scheduling, vectorization, and user-selected optimizations. These optimization types are described in the following subsections.

General optimizations
3.1

Efficient optimizations are very important to Cray Ada because the language inherently restricts certain types of vectorization and some movement of instructions. The general optimizations performed are the following:

- Subprogram inlining
- Common subexpression elimination
- Dead code elimination
- Local value propagation
- Range calculation and propagation
- Constant folding and propagation
- Check elimination
- Subprogram interface optimization
- Lifetime minimization
- Tail recursion elimination
- Dead store elimination

- Induction variable identification
- Copy propagation

The remainder of this subsection describes the specific optimizations that the optimizer performs. Only the potentially more significant optimizations are described. Other optimizations may be performed as appropriate, but they might not have as significant an impact on the execution efficiency of some Ada programs.

Call graph analysis optimizations 3.1.1

Call graph analysis optimizations are those based on the analysis of the call graph for a compilation unit or for a collection of compilation units. Complete call graph analysis is possible only when an entire program is optimized as a whole into an optimized collection. If this is not done, there will be at least some subprograms in the optimized collection for which no assumptions can be made regarding their potential callers. However, these subprograms will be those that belong only to the external interface of a unit within the collection. For subprograms that appear in a package body but not in its specification or for those nested within an other subprogram, only the calls that the optimizer can see can occur. Full knowledge of all possible calls can result in significant optimizations.

The types of optimizations possible from call graph analysis are inline expansion of subprograms and parameter substitutions.

Data flow analysis 3.1.2

Although not an optimization itself, data flow analysis is a key step in the optimization process. It is the foundation from which many of the subsequent optimizations are derived. The following are types of data flow analysis:

- Local data flow analysis. Local data flow analysis involves tracing data flow information within basic blocks. A *basic block* is a piece of code that has exactly one entry point and that executes sequentially without halts or branches. A basic block may have one or more exit paths associated with it.

- Common subexpressions. A *common subexpression* is a piece of code that appears more than once and computes the same value in each instance. At the source-code level for well-written programs, common subexpressions do not occur with any significant frequency. In translation of Ada to a lower-level intermediate representation, however, the compiler itself generates many instances of common subexpressions. They arise mainly in connection with address expressions for array and record accesses. By associating temporary variables, which are typically registers, with common subexpressions and replacing subsequent occurrences of the common subexpression with the temporary variable, considerable overall code improvement is possible.

- Range analysis. *Range analysis* is the tracking of the possible ranges in which expressions and variables are used in a program. Range analysis is used to substitute a constant value in place of an expression that can yield only that constant value. Range analysis is also used to drive other optimizations, including elimination of runtime checks and elimination of dead code.

- Constant folding. *Constant folding* is the replacing of an expression that always evaluates to a static literal value with the resulting literal value.

- Runtime check elimination and reduction. Runtime check elimination occurs when useless or redundant runtime checks are detected. Ada requires that the compiler insert rather extensive runtime checks for a variety of purposes if it cannot determine statically that the checks are redundant. Range analysis is used to detect runtime checks that can be eliminated or reduced. A check is reduced by being replaced with a simpler check. For example, a check to ensure that an integer variable is within a particular range could be replaced by a check on only the upper part of the range if it could be determined that the variable could not underflow the range. If a runtime check is determined statically to be true, it is replaced by a raise statement of the appropriate exception.

- Dead code elimination. *Dead code elimination* is the removal of portions of a subprogram that can never be executed or whose results will not be used. Range analysis, constant folding, and value propagation may cause the conditional part of an `IF` or `CASE` statement to evaluate to a constant value or to a value with a reduced range. This can result in the elimination of the portion of the `IF` or `CASE` statement that cannot be reached. Assignment to a variable that is not subsequently used results in the elimination of the source expression if it has no side effects.

- Value propagation. *Value propagation* is the replacement of a variable access with the expression that calculates the current value of the variable, whenever the expression is less expensive to evaluate than the variable access. A special case of value propagation is *constant propagation*, in which the expression is a literal value.

- Lifetime minimization. *Lifetime minimization* is the movement of an assignment to a variable as close as possible to the first use of the variable. Minimizing the span in which the variable is active can result in improved register utilization.

- Tail recursion elimination. *Tail recursion elimination* is the replacement of a recursive call at the end of a subprogram with a reassignment of parameter values and a jump back to the start of the subprogram. A tail recursive subprogram call is the last action performed during the subprogram invocation. For example, in the following program fragment, the recursive call on line 6 is tail recursive, but the recursive call on line 8 is not, because the constant `m` is added to the function result before returning from `F`.

```
1  function F (I: integer) return integer is
2  begin
3     if I = 0 then
4        return 1;
5     elsif I < 10 then
6        return F (I-1);
7     else
8        return m + F (I-1);
9     end if;
10 end F;
```

Tail recursion elimination can dramatically improve the running time of certain recursive algorithms by essentially turning them into iterative algorithms.

- Loop invariant code motion. *Loop invariant code motion* is the movement of expressions in a loop whose values do not change during the execution of the loop so that they occur outside of and before the loop. Loop invariant code motion is one of the two classic loop optimizations that often result in dramatic improvements in the running time of programs that make heavy use of looping constructs. Although user code seldom exhibits examples of invariant expressions at the source code level, the compiler often introduces loop invariant expressions into the lower-level intermediate representation, usually associated with addressing calculations.

- Induction variable identification. *Induction variables* are variables that change by a constant value on each iteration through a loop or are equal to a linear combination of other induction variables. After induction variables are detected, their usage can often be optimized (and sometimes entirely removed) by the replacement of expressions involving them with simpler expressions that calculate the same value (called *strength reduction*). In the following example, loop1 can be replaced by loop2, which involves less costly arithmetic operations:

```
i := 1;
loop1: while i < 10 loop
          j := i * 5;
          a[j] := b[j];
          i := i + 1;
       end loop;
```

```
j := 5;
loop2: while j < 50 loop
          a[j] := b[j]
          j := j + 5;
       end loop;
```

- Dead store elimination. *Dead store elimination* is the removal of assignments to variables that are never used after they are assigned.

- Copy propagation. *Copy propagation* is the elimination of copy statements in a program. Copy statements are statements of the form `A := B`. The copy is propagated by the replacement of subsequent uses of `A` by `B`. Copy propagation also works on statements of the form
  `A := <literal value>`.

Inlining code
3.1.3

The inlining of subroutines can greatly decrease the overhead associated with making subroutine and function calls.

Whenever possible, calls to subprograms are inlined automatically or when such calls are flagged with pragma `INLINE`. In addition, any calls to noninterface subprograms that are called from only one place are inlined. Finally, cal'3 to subprograms having bodies small enough that the inlining is cheaper in code space than the corresponding call are inlined. The inlining of subprograms called from only one place can be suppressed to preserve a simple mapping between source subprograms and object code.

Small subprograms fall into two categories, depending on whether or not they belong to the external interface of the unit in which they appear. If a small subprogram does not belong to the external interface of the optimized unit, a body is not generated for it, and all calls to it are inlined.

When a small subprogram belongs to the external interface of the optimized unit, a body is generated for it to support external calls. However, calls within the unit are inlined. Even when the subprogram is part of the external interface of a unit, if that unit is a hidden unit within a collection, generation of the body is suppressed.

Instruction scheduling
3.2

Cray Ada includes an instruction scheduler, which reorders generated instructions such that they execute faster on Cray Research hardware than they would have otherwise. Because Cray Research hardware can perform program execution in parallel both across and within functional units (*pipelining*), instruction scheduling can significantly increase the speed at which programs execute.

Additionally, movement of load instructions to minimize wait times due to the speed of memory access relative to other CPU operations can provide substantial gains in performance, particularly in CRAY-2 systems. On CRAY Y-MP and CRAY X-MP systems, further improvements in performance may be realized by the chaining of vector operations.

Vectorization 3.3

Vectorization is the process of changing a loop to an equivalent form which operates on several iterations of the loop in parallel. This is made possible by special hardware support on Cray Research systems computers, and is the largest single execution performance improvement of any optimization available.

Cray Ada does not have a mechanism for directly expressing vector operations, so the compiler is capable of automatically transforming sections of code from scalar to equivalent vector operations. The following three primary criteria are used in determining whether a construct will vectorize:

- The vectorized code meets the requirements for generated Ada code described in the Ada Language Reference Manual (LRM).
- The vectorized program produces the same results as does the scalar version.
- The optimization cost when weighted against the performance improvements shows the vector support to be important.

The Cray Ada compiler has a mechanism for providing detailed information about vectorization opportunities in compiled code. This feature is available through the `-T` option on the `ada` command line. Cray Ada also supports a system-dependent pragma, `VECTORIZE_LOOP`, which allows for user control of vectorization. For more information on this pragma, see *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

For your reference, the following terms are used in this subsection:

- An *induction variable* is a variable that is incremented or decremented with each iteration of a loop. The increment or decrement may be accomplished by +, -, or *. Division is not supported. Operations of the form : `J := 10 - J;` are not legal for induction variables because of the switching of the sign of `J` from iteration to iteration of a loop.
- A *dependence* is an ordering relationship between operators that describes the order in which the operation must execute to get the correct results. Vectorization must preserve this ordering.
- A *recurrence* is a situation in which two or more operations are in dependence with one another. (They depend on one another for the results of their respective operations.) The vectorization of recurrences can occur only under special conditions.
- A *vector array reference* is the usage of an array with a subscript that is variant in the loop. Examples of such subscripts are induction variables or linear functions of induction variables.
- An *invariant* is an object or constant that is used but not defined in the loop.
- A *scalar temporary* is a variable defined in the loop and later referenced in the loop.

The discussion in the following subsections defines vectorization as it applies to Cray Ada release 2.0. Ada constructs that form vectorizable code sequences and those that inhibit vectorization are defined. The concepts and terms of vectorization are defined. Examples of Ada code are used throughout to illustrate and explain. At points, individual statements are labeled with `si`, in which `i` is a digit for the purpose of identifying particular statements for later discussion.

Concepts of vectorization 3.3.1

An understanding of vectorization includes not only the forms of Cray Ada code that may vectorize but underlying concepts as well. The following subsections are an introduction to those concepts.

Vectorization of induction variables 3.3.1.1

An *induction variable* is a variable defined in a loop as a function of the loop index variable and the loop's iterations. For example, k, j, and l are induction variables:

```
FOR i IN 1..100 LOOP
  k := i + 2;
  j := j + 1;
  l := k * 1;
END LOOP;
```

k is defined in terms of i, j is defined by the loop iterations, and l is derived from k.

j in this example is a special case of an induction variable called a *constant increment integer* (CII). These are variables that are defined solely as an increment or decrement of themselves and as constant values.

Definitions and uses 3.3.1.1.1

To allow vectorization to occur, certain restrictions are imposed upon the assignments and uses of induction variables.

Whenever an induction variable is defined in the loop before it is used, vectorization can occur.

Example:

```
FOR i IN 1..100 LOOP
  j := j + 2;
  a(j) := b(i) * 10;
  k := i * 2;
  c(k) := b(j) * 20;
END LOOP;
```

The range of values for j and k can be determined as a function of the loop index variable i, and their definitions and uses can be vectorized.

However, when an induction variable such as k is used before it is defined (swap the third and fourth statements shown previously), the loop is not vectorizable. The first iteration of the loop would have to be executed separately to account for the initial value of k.

For CII variables such as `j`, vectorization will occur regardless of whether they are used before they are defined.

Other restrictions 3.3.1.1.2

Other restrictions are the following:

- Induction variables must be integers.
- A CII can be self-incrementing or self-decrementing with + and –.

  The CII variable must be on the left-hand side of a subtraction sign. For example, the following is not a valid CII.

  ```
          j := 10 - j;
  ```

- Induction variables cannot be assigned to an `IF` or a `CASE` construct.

Data dependence 3.3.1.2

Data dependence is an ordering relationship between two statements that use or produce the same data. This ordering must be preserved in order to generate the same results. The analysis of data dependence determines whether the statements can be vectorized without violating this ordering.

This section first describes the various types of dependence relationships between two statements. Following this is a discussion of the ways these relationships affect the vectorization of a given loop.

Dependence types 3.3.1.3

The types of dependence are flow dependence, antidependence, output dependence, and unknown dependence. These dependence types describe the relationship between two statements and a single variable (scalar or array reference), and how that relationship affects the execution of the loop.

Flow dependence 3.3.1.3.1

Statement `s2` is *flow dependent* on statement `s1` if an execution path exists from `s1` to `s2` and the same variable (scalar or array element) is assigned in statement `s1` and used in statement `s2`.

Example 1:

```
     FOR i IN 1..100 LOOP
  s1:  a(i) := b(i) * 5;
  s2:  c(i) := a(i) + 3;
     END LOOP;
```

In this example, `s2` is flow dependent on statement `s1`. The order of these two statements must be preserved so that the value of `a(i)` is produced before it is used. Vectorization of this loop does not violate this ordering.

Table 2, is a work tableau showing the execution of the iterations of this loop. The columns indicate the loop iterations as the loop index variable increases, and the rows show the iterations in which the statements indicated define or use a value. Each arrow points from a statement that defines a value to a statement that uses that value.

Table 2. Example 1 work tableau

i = 1	i = 2	i = 3	... i = 100
s1	s1	s1	... s1
↓	↓	↓	↓
s2	s2	s2	... s2

The vertical arrows indicate that the value computed in the given iteration of the loop is used by `s2` in the same iteration. This is true in Example 2.

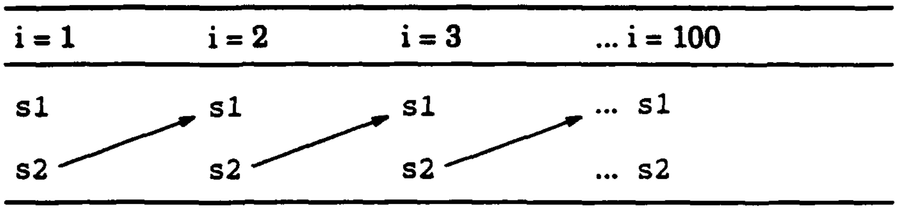
Example 2:

```
   FOR i IN 1..100 LOOP
s1: a(i) := b(i) * 5;
s2: b(i+1) := c(i) + 3;
   END LOOP;
```

In example 2, `s1` is flow-dependent on `s2` across an iteration of the loop. The value of `b(i+1)` must be computed in each iteration before it is used in the next iteration. Therefore, this loop cannot be vectorized as it is written. A transformation called ***statement reordering*** can be performed to switch the order of the statements and allow vectorization to occur. This transformation can automatically be done by vectorization.

The work tableau in Table 3 shows why the loop does not have independent iterations.

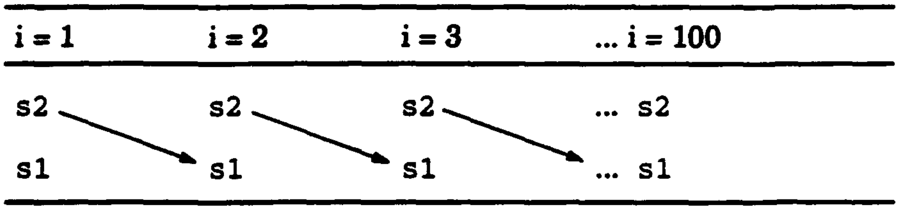
Table 3. Example 2 work tableau



i = 1	i = 2	i = 3	... i = 100
`s1`	`s1`	`s1`	... `s1`
`s2`	`s2`	`s2`	... `s2`

The diagonal arrows show that a cross-iteration dependence exists because the values set by `s2` are not used until the next iteration. For cases such as this, a compiler automatically reorders statements `s1` and `s2`, resulting in the following tableau in Table 4.

Table 4. Example 2 reordered work tableau



i = 1	i = 2	i = 3	... i = 100
`s2`	`s2`	`s2`	... `s2`
`s1`	`s1`	`s1`	... `s1`

Although the cross-iteration dependence still exists from `s2` to `s1`, the loop is now vectorizable because all of the computations for `b` performed in iterations 1 through 100 will be done before (hence, the downward direction of the arrows) those values of `b` are used in statement `s1`.

Anti dependence
3.3.1.3.2

Statement `s2` is *anti dependent* on statement `s1` if an execution path exists from `s1` to `s2` and if the same variable (scalar or array element) is used by `s1` and assigned by `s2`.

Example 3:

```
  FOR i IN 1..100 LOOP
s1: a(i) := b(i) * 5;
s2: b(i) := c(i) + 3;
  END LOOP;
```

In example 3, s2 is anti dependent on s1. The order of these two statements must be preserved so that the value of b(i) is used before it is redefined. Vectorization does not violate this ordering. This loop has the same execution tableau as does Table 2, page 29.

Example 4:

```
   FOR i In 1..100 LOOP
s1: a(i) := 5;
s2: b(i) := a(i+1) + 3;
   END LOOP;
```

Example 4 shows that s1 is antidependent on s2 across the iterations of the loop. The loop can be vectorized only if the statements are reordered in a way similar to that as in Example 2 and Table 3, page 30 and Table 4, page 30. The statement reordering can be done automatically here as well.

Output dependence 3.3.1.3.3

Statement s2 is *output dependent* on statement s1 if an execution path exists from s1 to s2 and if the same variable (scalar or array element) is assigned in both statements.

Example 5:

```
   FOR i In 1..100 LOOP
s1: b(i) := 5;
s2: b(i) := a(i+1) + 3;
   END LOOP;
```

In example 5, s2 is output dependent on s1. That is, the value computed for b(i) in s1 is recomputed (overwritten) in s2. This type of dependence has the same properties as flow and antidependence in terms of cross-iteration dependencies and statement reordering.

Unknown dependence 3.3.1.3.4

Sometimes the dependence relationship between statements cannot be determined. This can happen in cases such as the following:

- A subscript is subscripted (indirect addressing).
- The subscript does not contain the loop variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is nonlinear with respect to the loop variable.

When one of these cases occurs, the compiler assumes that a dependence exists and will not vectorize the loop. However, if you know the code well enough to know that a dependence cannot exist, inserting a pragma `VECTORIZE_LOOP` will allow vectorization to occur.

Recurrence 3.3.1.3.5

Recurrences are the result of data dependence cycles between statements and variables in the loop. This occurs when one or more statements are in dependence with each other. For example:

Example 6:

```
 FOR i In 1..100 LOOP
s1: a(i) := b(i-1) * 5;
s2: b(i) := a(i) + 3;
 END LOOP;
```

In example 6, `s2` depends on `s1` for `a(i)`, and `s1` depends on `s2` for `b(i)`. This loop is not vectorizable because `s1` uses values of `b` calculated on previous iterations of the loop. The execution tableau for the loop with respect to `b` is the same as that in Table 3, page 30, and Table 4, page 30, showed that statement reordering could remove this dependence. However, that is not the case here, because there is also the dependence from `s1` to `s2` for `a(i)`. This dependence prohibits reordering because doing so would cause incorrect values of `a(i)` to be used in `s2`. This is called a *multistatement recurrence or cycle*.

Another type of recurrence involves a single statement and one variable.

Example 7:

```
     FOR i IN 1..100 LOOP
s1: a(i + 1) := a(i) *5;
    END LOOP;
```

Here, there are two dependencies: an antidependence from a(i) to a(i + 1) in the same iteration of the loop, and from a(i + 1) to a(i) across an iteration of the loop. This is called a *single-statement recurrence* and is not vectorizable.

Recurrences also occur for scalar variables.

Example 8:

```
     FOR i IN 1..100 LOOP
s1: a(i) := b(i) * x;
s2: x:= a(i) * 10;
     END LOOP;
```

Example 8 has the same properties as that of example 6, with the cross-iteration dependence of s1 on s2 with x. This loop is not vectorizable. A situation similar to that in example 7 occurs with scalars for a single statement recurrence. For example:

Example 9:

```
     FOR i IN 1..100 LOOP
s1: x := x * a(i);
     END LOOP;
```

The same dependencies exist in example 8 as in example 7 with respect to x. There is a difference in example 9, however, because this is actually a vectorizable reduction.

Certain recurrences involving array references are vectorized by the Cray Ada compiler.

Example 10:

```
     FOR i IN 1..100 LOOP
s1: a(i +10) := a(i) * b(i);
     END LOOP;
```

The recurrence in example 10 begins on the eleventh iteration and every iteration subsequent to that. The eleventh iteration is when the value of a(i) will be that which was computed on the first iteration. The first ten iterations are vectorizable, however, because values computed there are not used on subsequent iterations up to the eleventh. This sort of partial vectorization occurs for recurrences whenever at least three iterations are proven to be independent.

Vectorizations performed by Cray Ada 3.3.2

The Cray Ada 2.0 compiler performs a subset of the possible vectorizations. The following is a list of the kinds of Ada constructs that the Cray Ada vectorizer supports. These are explained further in the following subsections.

- FOR, while and loop loops
- Float, integer, Boolean, and enumerated types and subtypes
- Single and multidimensional arrays
- Most Ada operators
- Reduction operations
- Single-level IF statements within loops
- Single-level CASE statements within loops
- Sequence operations

Vectorization loop forms 3.3.2.1

Loops that vectorize must be of suitable form. The rules for constructing loops with these forms and examples are defined in the following subsections.

Loop exiting 3.3.2.1.1

Vectorizable loops may not be exited by any method other than those described in the following subsections. Loops with multiple exits are not vectorized with the Cray Ada 2.0 compiler.

FOR loops 3.3.2.1.2

FOR loops with either forward or reverse directions are supported. The loop parameter or index of a FOR loop is restricted for enumerated types.

The discrete range of the loop parameter must not be discontinuous. For Cray Ada 2.0, the enumerated type is restricted from being defined by an enumeration representation clause.

Example:

```
Type Codes IS (A, B, C, D, E, F);
FOR Codes USE
  (A = > 1, B = > 5,C=> 6, D=8, E=>9, F=>10);
FOR Code IN Codes LOOP
. . .

End Loop
```

In this example, Code would successively take on the values 1, 5, 6, 8, 9, and 10. Values 2 through 4 and 7 would be unused. The gaps in the range cause the loop to be not vectorized.

```
A : array (INTEGER range 1 .. 100) of float;
B : array (INTEGER range 1 .. 50) of float;
I, J : Integer;
Inc : Integer;

. . .

For I in 1 .. 50 loop   —Forward step
  A(I) := B(I);
End Loop;

For I in Reverse 1 .. 50 loop   —Reverse step
  A(I) := B(I);
End Loop;
```

while loops 3.3.2.1.3

Both fixed and variable increment loops vectorize. The form of the while condition is restricted to the following forms:

<expression> <test> <index variable>

<index variable> <test> <expression>

The *<expression>* must be loop invariant. The *<test>* may be any relational operator.

The loop increment must be the last statement in the loop and one of the following:

<index variable> : = *<index variable>* + *<increment expression>*

<index variable> := *<index variable>* – *<increment expression>*

The *<increment expression>* must be loop invariant. The *<index variable>* must be invariant within the loop.

The following are examples of vectorizable `while` loops:

```
A : array (INTEGER range 1 .. 100) of float;
B : array (INTEGER range 1 .. 50) of float;
I, J : Integer;
Inc : Integer;

. . .

While I > 10 loop
  A(I) := B(I);
  I := I - 2;     —Fixed increment WHILE loop
end loop;

While J <= 50 loop
  A(J) := B(J);
  J := J  + Inc;  —Variable increment WHILE loop
end loop;
```

`loop` loops
3.3.2.1.4

Both fixed and variable increment loops vectorize. The following are `loop` form requirements.

The loop index increment or decrement statement must either precede directly or follow the exit statement.

Only one exit may be present in the loop. The exit must be either the last statement in the loop or the next-to-last statement followed by the increment or decrement of the loop index. The exit statement must be in one of the following two forms:

```
IF <expression> THEN
  EXIT
END IF;


EXIT WHEN <expression>;
```

The form of the `loop` condition is restricted to the following:

<expression> <test> <index variable>

<index variable> <test> <expression>

The following are `loop` loop examples:

```
I := 40;
LOOP
  A(I) := B(I);
  I := I-3;
  EXIT WHEN I <= 0;
END LOOP;
```

```
J := 2;
LOOP
  A(J) := B(J)
  J := J + Inc;
  IF J > 20 THEN
    EXIT;
  END IF;
END LOOP;
```

Vectorization constructs in loops 3.3.2.2

A loop that has one of the allowable vectorizable forms may vectorize if the contents of the loop are themselves vectorizable. The vectorizable Ada statement patterns for Cray Ada 2.0 are defined in this section.

`IF` statement 3.3.2.2.1

Simple single-level `If/Then/Elsif/Else` constructs will vectorize if the construct is within a vectorizable loop structure. In the following example, there are no dependencies in the use and assignments, so the `if` statement will vectorize:

```
A : array (INTEGER range 1 .. 100) of float;
B : array (INTEGER range 1 .. 100) of float;

. . .

For I in 1 .. 100 loop
  If A(I) > 0.0 then
    A(I) := A(I) / B(I);
  Elsif A(I) < 0.0 then
    A(I) := A(I) - 1;
  Else
    A(I) := -1.0;
  End if;
End loop;
```

CASE statement 3.3.2.2.2

Simple single-level CASE statement will vectorize if the statement is within a vectorizable loop structure. In the following example, there are no dependencies in the use and assignments, so the CASE statement will vectorize:

```
A : array (INTEGER range 1 .. 100) of float;
B : array (INTEGER range 1 .. 100) of float;
C : array (INTEGER range 1 .. 100) of float;

. . .

For I in 3 .. 63 loop
  Case I is
    When 6 =>
      A(I) := B(I):
    When 8 =>;
      A(I) := 12*B(I):
    When Others =>
    A(I) := C(I);
  End Case;
End loop;
```

EXIT statement 3.3.2.2.3

The EXIT statement may occur only as previously discussed for the allowable loop forms.

Reductions 3.3.2.2.4

Reductions are a special class of computations that *reduce* an array of values to a single scalar result. These computations can be vectorized under certain conditions.

The following are examples of vectorizable reductions:

Example 1:

```
 FOR i IN 1..100 LOOP
s1:  x:= x + a(i);
s2:  y:= (b(i) * d(i)) *y;
s3:  z:= z - c(i);
s4:  x1 := x1 / e(i);
 END LOOP;
```

Arithmetic reduction operations involving integer and floating-point addition, subtraction, multiplication, and floating-point division are all vectorizable. For subtraction and division, however, the scalar reduction variable must be on the left-hand side of the - or /.

For example, the following will not be vectorized. In addition, reductions are only supported for floating-point data types.

Example 2:

```
 FOR i IN 1..100 LOOP
s1: x := a(i) - x;
s2: y := b(i) / y;
 END LOOP;
```

The reduction variable must be a scalar variable or an invariant array reference such as an array with a constant subscript, as in the following:

```
 FOR s1: a(1) := a(1) * b(i);
s2: a(c) := a(c) + b(i); –c is a constant
 END LOOP;
```

The expression from which the result is reduced can be arbitrarily complex.

Types of reductions 3.3.2.2.5

The base types of vectorization reductions are integer, float, and Boolean. Examples of Boolean reductions are the following:

```
  For i IN 1..100 LOOP
s1: x:= x AND Bool_A(i)
s2: y:= y OR (Bool_A(i) AND Bool_B(i);
s3: z:=z XOR Bool_B(i);
  END LOOP;
```

Reductions involving searches for the minimum or maximum value of an array are vectorizable as well, as shown in the following examples:

```
FOR i IN 1..100 LOOP
  IF s < a(i) THEN
    s := a(i);
  END IF;
END LOOP;
```

```
FOR i IN 1..100 LOOP
  IF t > a(i) THEN
    t := a(i);
  END IF;
END LOOP
```

The first loop represents a *max* reduction and the second loop is a *min* reduction where, at the end of the loop, the scalar variable will contain the maximum or minimum value of the array.

Min/Max reductions can be vectorized only when the data types are integer or float, and when the IF statement contains no other code besides the reduction assignment statement. The reduction variable can be an invariant array reference as with other reductions.

Expression elements 3.3.2.3

Some of the allowable elements of expressions that may appear in vectorizable Cray Ada constructs are described in the following subsections.

Array references 3.3.2.3.1

The closer the compiler comes to knowing the range of values an object may take on, the better the possibility of vectorization. This is particularly true in index expressions. The following is an example:

```
FOR Index In 1..100 LOOP
  Offset := Offset + 1;
  A(Index) := B(2*Index + 3*Offset - 2);
END LOOP;
```

Operators 3.3.2.3.2

All of the logical, relational and binary adding operators are supported, with the exception of &. The highest-precedence operators are supported as well.

Gather/scatter operations 3.3.2.3.3

Gather/scatter operations are those involving nonstep-wise access to an array. These operations are supported as long as overlap can not be found in the array references. For example, the following loop will automatically vectorize because there are no dependencies:

```
A : array (INTEGER range 1 .. 100) of float;
B : array (INTEGER range 1 .. 100) of float;
C : array (INTEGER range 1 .. 100) of float;
D : array (INTEGER range 1 .. 100) of float;

. . .

For I in 1 .. 100 loop
  A(I) := B(D(I));       —Gather
  C(D(I)) := B(I);       —Scatter
END LOOP;

For I in 1 .. 100 loop
  A(I) := B(I**4);       —Gather
  C(I/2) := B(I);        —Scatter
End Loop;
```

If the compiler cannot prove that no dependency exists, it will not vectorize the loop. In these cases, if you are certain that no dependency exists, pragma VECTORIZE_LOOP can be used immediately before the loop to attempt to force vectorization. See page 65 for more information on pragma VECTORIZE_LOOP. An example in which this pragma is required for the loop to vectorize is the following:

```
A : array (INTEGER range 1 .. 100) of float;
B : array (INTEGER range 1 .. 100) of integer;
C : array (INTEGER range 1 .. 100) of integer;

. . .

Pragma VECTORIZE_LOOP(on);
For I in 1 .. 100 loop
  A(B(I)) := A(C(I);   —This will vectorize because of
End Loop;               —VECTORIZE_LOOP Pragma
```

The VECTORIZE_LOOP pragma affects only the loop immediately following it.

Vectorizable scalar types 3.3.2.4

Most objects defined as Ada types integer, float, Boolean, and character are allowable for vectorization. Variables defined as an enumerated type are also vectorizable.

Vectorizable array types 3.3.2.5

Arrays whose components are of a vectorizable scalar type can be vectorized.

Vectorizable record types 3.3.2.6

Records, discriminated or not, whose referenced components are of a vectorizable scalar or array type are vectorizable. However, only the final record referenced in a given record component qualification may be discriminated.

Example:

```
TYPE Ar IS ARRAY (1..100) OF FLOAT
TYPE R1 (S : Boolean := True) IS
  RECORD
    CASE S IS
      WHEN True =>
        A : Ar;
      WHEN False =>
        B : Integer;
    END CASE;
  END RECORD
TYPE R2 IS
  RECORD
     C : Ar;
     D : R1;
  END RECORD
V1 : R1;
V2 : R2;

...

FOR I In 1..100 LOOP
  V1.A(I) := V2.D.A(I);
END LOOP;
```

Vectorizable library interfaces 3.3.2.7

As a general rule, subprogram calls in loop bodies inhibit vectorization. However, certain of the subprograms described in "Library Interface," page 119, are available in vector versions, and thus, loops containing calls to these routines can still be vectorized as long as no other vectorization inhibitors are present.

Vectorizable functions are the following:

- Trigonometric functions, which include the following:

  `Sin, Cos, Tan, Cot, Asin, Acos, Atan, Atan2`

- Hyperbolic trigonometric functions, which include the following:

  `Sinh, Cosh, Tanh`

- Logarithmic functions, which include the following:

  ```
  Log, Log10, **, Exp, Sqrt
  ```

- Boolean array routines, which include the following:

  ```
  Leadz, Popcnt, Shiftl, Shiftr
  ```

- Miscellaneous arithmetic functions, which include the following:

  ```
  Sign, Trunc, Ranf
  ```

The vector versions of these routines yield the same results as do the scalar versions, except that under certain circumstances, the vector version of Ranf may produce pseudo-random numbers in a different order than would the scalar Ranf. Programs that might be sensitive to this should be written and compiled in such a way that they always use the same version of Ranf (scalar or vector).

Constructs that do not vectorize
3.3.3

The following subsections describe Ada language constructs that inhibit vectorization. This is not an inclusive list, but it provides guidelines to assist you in writing vectorizable code. The best way to determine whether a specific structure will vectorize is to compile the source code, using list options that provide details on the constructs that vectorized.

The subsections contain information on the constructs that will probably always inhibit vectorization along with those constructs that could, in theory, be vectorized but are not in Cray Ada 2.0. For many of the constructs that can be theoretically vectorized, information is provided to guide users in revising their code to allow vectorization with Cray Ada 2.0.

Using pragma Vectorize_Loop has no effect on most loops described here. This pragma can be used to tell the compiler only that potential dependencies are not true dependencies.

String variables, constants, and operations
3.3.3.1

Using strings in any context inside a loop will inhibit vectorization. For example, the following will not vectorize:

```
S : String(1..10);

. . .

FOR I IN 1..10 LOOP
  S(I) := '0';          —This will not vectorize
END LOOP;
```

Fixed-point variables, constants, and operations 3.3.3.2

Loops containing fixed-point variables, constants, or operations inhibit vectorization.

Example:

```
TYPE F IS DELTA .01 RANGE .0 .. 1.0;
A, B : ARRAY(1..10) OF F;

. . .

FOR I IN 1..10 LOOP
  A(I) := A(I) + B(I);
END LOOP;
```

On Cray Research systems, fixed-point arithmetic is always much slower than either integer or floating-point arithmetic. Converting fixed-point types to floating-point subtypes makes vectorization possible. This is true for only multiplication and division, and a loss of accuracy will result.

Access type variables, constants, and operations 3.3.3.3

Loops containing access-type variables, constants, or operations inhibit vectorization.

Example:

```
TYPE ACC IS ACCESS INTEGER;
A, B: ARRAY (1 ...100) OF ACC;
...
FOR I IN 1...100 LOOP
  A(I) := B(I);
END LOOP;
```

Array types with nonvectorizable components 3.3.3.4

Arrays with components of types record, array, or nonvectorizable data types are not vectorizable.

Record types with nonvectorizable components 3.3.3.5

Records whose referenced components are not of a vectorizable data type can not be vectorized. Record references whose qualification path includes more than one discriminated record, or that contain a discriminated record that is not the final record referenced in the path, are not vectorizable.

Example:

```
TYPE R0 (R0_Kind : INTEGER) IS
     RECORD
       a0 : INTEGER;
       CASE Kind
         WHEN 1 =>
           a1 : Array_Type1;
         WHEN 2 =>
           a2 : Array_Type2;
         WHEN OTHERS =>
           NULL;
         END CASE;
     END RECORD;

TYPE R1 (R1_Kind : INTEGER) IS
     RECORD
       CASE Kind
         WHEN 1 =>
           a : R0(R0_Kind => 1);
         WHEN 2 =>
           b : R0(R0_Kind => 2);
         WHEN OTHERS =>
           NULL;
       END CASE;
     END RECORD;

TYPE R2 IS
     RECORD
       c : INTEGER;
       d : R0 (Kind => 1);
     END RECORD;
v1 : R1(Kind => 2);
v2 : R2;

..............

FOR in IN 1..100 LOOP
     v2.d.a.a1(i) := v1.b.a2(i);
END LOOP;
```

The previous loop is not vectorizable because there are two discriminated records referenced in the qualification path (`d.a` and `v1.b`).

Another limitation in the vectorization of records involves multiple unconstrained or dynamically allocated components in a single record. This includes combinations of unconstrained arrays and discriminated record components whose sizes cannot be determined at compilation. Whenever a record component that follows an indeterminably-sized component in the record-type specification is referenced, vectorization of that reference cannot occur.

Example:

```
PROCEDURE Example (Size : IN INTEGER) IS
   TYPE Array_Type1 IS ARRAY (INTEGER RANGE<>) OF INTEGER;
   TYPE R0 (R0_Kind : INTEGER ) IS
    RECORD
     a0 : INTEGER;
     CASE Kind
     WHEN 1 =>
      a1 : Array_Type1(1..Size):
     WHEN 2 =>
      a2 : Array_Type1(1..Size);
     WHEN OTHERS =>
      NULL;
    END CASE;
   END RECORD;

   TYPE R1 IS
    RECORD
     b0 : INTEGER
     b1 : R0 (R0_Kind => 1);
     b2 : ARRAY (1..100) OF INTEGER;
    END RECORD;

   TYPE R2 IS
    RECORD
     c0 : INTEGER
     c1 : Array_Type1(1..Size)
     c2 : R0 (Kind => 1);
    END RECORD;

BEGIN
 ........
END EXAMPLE;
```

References to components b2 of record R1 and c2 of record R2 will not be vectorized because the size of components b1 and c1, respectively, are not known at compile time. However, references to components b0 and b1, and c0 and c1 are vectorizable.

Packed data 3.3.3.6

The use of pragma PACK or explicit packing of data by representation specifications inhibits the vectorization of any object of that type. The example that follows will not vectorize:

```
SUBTYPE Byte IS Integer RANGE 0..255;
TYPE Byte_Array IS ARRAY (1..8) OF Byte;
PRAGMA Pack(Byte_Array);
Byte_Obj: Byte_Array;
```

```
FOR I IN 1..8 LOOP
  Byte_Obj(I) := Byte_Obj(I) MOD 4;
END LOOP;
```

Procedure and function calls 3.3.3.7

Procedure and function calls that are not inlined inhibit vectorization. The only exceptions to this are most Cray Research vectorizable library interface routines and most predefined Ada attribute functions that are fully vectorizable. The vectorization inhibitors include the following:

- Explicit function or procedure calls that are not inlined
- Ada tasking operations
- Ada I/O package routines
- Ada allocators
- Some Ada attributes
- Operations on some composite types

Examples of each of these vectorization inhibitors follow.

Explicit function and procedure calls 3.3.3.7.1

The following example will not vectorize because of the procedure call.

Procedure Proc1 is used in the following:

```
SUBTYPE Ar_Index IS Positive RANGE 1..65;
  SUBTYPE Ar_Value IS Ar_Index RANGE 1..64;
  TYPE Ar IS ARRAY (Ar_Index) OF Positive;

  A : Ar;

  PROCEDURE Proc2(I: IN Ar_Value) IS
    B :Ar;
  BEGIN
    FOR Index2 IN 1..I LOOP
      B(Index2) := B(Index2 + 1);
    END LOOP;
  END Proc2;

  PROCEDURE Proc3(I : IN Ar_Value) IS
  BEGIN
    A(I) := A(I + 1);
  END Proc3;

BEGIN
  FOR Index1 IN 1..63 LOOP
    Proc2(Index1);
  END LOOP;
  FOR Index1 IN 1..63 LOOP
    Proc3(Index1);
  END LOOP;
  Proc2(22);
END Proc1;
```

The first loop in `Proc1` will not vectorize because of the presence of the call to `Proc2` within the loop. The body of `Proc2` contains a loop that will vectorize. The second loop in `Proc1` vectorizes because of the automatic inlining of the call to `Proc3`. Finally, the call to `Proc2` at the end of `Proc1` has no effect on vectorization other than the prohibiting of the inlining of `Proc2` in the first loop in `Proc1`. This is true unless an explicit request was made on the command line to inline procedures or unless a pragma `INLINE` is used.

Ada tasking operations 3.3.3.7.2

Ada tasking operations that appear in the body of a loop result in procedure and/or function calls. The appearance of such calls inhibits vectorization.

Ada allocators 3.3.3.7.4

The use of allocators in a loop inhibits vectorization because they result in procedure and/or function calls to runtime support routines.

Example:

```
I_Acc : ACCESS Integer;
A : array (INTEGER range 1 .. 100) of integer;

. . .

FOR I IN 2..12 LOOP
  I_Acc := NEW Integer;
  . . .
  A(I) := I_Acc.All;
END LOOP;
```

Ada attributes 3.3.3.7.5

Using some attributes results in the calling of procedures and/or functions. These calls inhibit vectorization. Attributes that are functions are defined as such in Appendix A of the *Ada Language Reference Manual* (LRM).

Example:

```
TYPE Codes IS (A,B,C,D,E,F);
FOR Codes USE (A=> 1,B=> 5,C=> 6,D=> 320,E=> 321,F=>322);
```

```
I : Integer;
S : Codes;
A : array (INTEGER range 1 .. 100) of integer;

  . . .

FOR J IN 2..100 LOOP
  S := Integer'Pred(I);   —This attribute is a function
  . . .
  A(J) := 0;
END LOOP;
```

Ada allocators 3.3.3.7.4

The use of allocators in a loop inhibits vectorization because they result in procedure and/or function calls to runtime support routines.

Example:

```
I_Acc : ACCESS Integer;
A : array (INTEGER range 1 .. 100) of integer;

. . .

FOR I IN 2..12 LOOP
  I_Acc := NEW Integer;
  . . .
  A(I) := I_Acc.All;
END LOOP;
```

Ada attributes 3.3.3.7.5

Using some attributes results in the calling of procedures and/or functions. These calls inhibit vectorization. Attributes that are functions are defined as such in Appendix A of the LRM.

Example:

```
TYPE Codes IS (A,B,C,D,E,F);
FOR Codes USE (A=> 1,B=> 5,C=> 6,D=> 320,E=> 321,F=>322);
```

```
I : Integer;
S : Codes;
A : array (INTEGER range 1 .. 100) of integer;

  . . .

FOR J IN 2..100 LOOP
  S := Integer'Pred(I);   —This attribute is a function
  . . .
  A(J) := 0;
END LOOP;
```

Most attribute usages do not result in calls to procedures or functions. Cases that result in calls are described in the remainder of this subsection.

Most attribute usages do not result in calls to procedures or functions. Cases that result in calls are described in the remainder of this subsection.

- The following is a type attribute and its description:

Attribute	Description
Size	Involves a function call in the case of objects with a nonstatic discriminated record subtype having at least two discriminants or having a single discriminant with a very large or nonstatic range.

- The following are discrete type attributes and their descriptions:

Attribute	Description
Image	A string returning function.
Value	A function complementary to `IMAGE`, with a string parameter.
Pos, *Pred* and *Succ*	These attributes are implemented by function calls only when applied to an enumeration type with an enumeration representation clause and when the enumeration codes span a range of more than 256 values. *Pos* is used implicitly in places such as `FOR` loops and array indexing.
Width	Always implemented with a function call.
Extended attributes	Which include the following: `EXTENDED_IMAGE`, `EXTENDED_VALUE`, `EXTENDED_DIGITS`, `EXTENDED_FORE`, and `EXTENDED_AFT` are always implemented with function calls.

- The following are task attributes and their descriptions:

Attribute	Description
Callable, *count*, and *terminated*	These three attributes are implemented as calls to functions.

- The following are fixed-point attributes and their description:

Attribute	Description
Fore, *mantissa*, and *large*	These attributes are implemented as function calls only when applied to a nonstatic fixed-point subtype.

Exceptions raised within loops 3.3.3.8

Any loop in which exceptions may be raised, whether predefined by Ada implementation defined, or user defined, inhibits vectorization.

Example:

```
A : array (INTEGER range 1 .. 100) of integer;
B : Integer;

. . .

For I in 1 .. 100 loop
  If A(I) = 0 then
    Raise ZERO_VAL; —User-defined exception inhibits vectorization
  else
    A(I) := B / A(I);
  end if;
end loop;
```

Pragma SUPPRESS or pragma SUPPRESS_ALL may be used to suppress all exceptions not explicitly raised by a RAISE statement and allow for constructs with predefined exceptions to vectorize.

With optimization enabled, the compiler attempts to remove as many of the predefined exception checks as it determines it is safe to do. However, it cannot generally remove all the checks from the loop. To ensure that automatic checks are not included in the loop, use pragma SUPPRESS or pragma SUPPRESS_ALL inside your code. Alternatively, the Ada command-line option -i can be used to suppress runtime checks. A loop will not be vectorized if any checks exist in the loop.

Example:

```
A : array (INTEGER range 1 .. 100) of integer;

. . .

FOR I IN 2..100 LOOP
  A(I) := A(I - 1);    —Vectorization inhibited
END LOOP;
```

You could rewrite the preceding loop so that it would vectorize, as follows:

```
A : array (INTEGER range 1 .. 100) of integer;
K : Integer;
. . .
K := A(1);
FOR I IN 2..100 LOOP
  A(I) := K;    —Vectorization allowed
END LOOP;
```

The following loop cannot be rewritten to support vectorization:

```
A : array (INTEGER range 1 .. 100) of integer;
B : array (INTEGER range 1 .. 100) of integer;
. . .

FOR I IN 2..100 LOOP
  A(I) := A(I) - A(I - 1) * B(I);
                                   —Vectorization
                                     inhibited
END LOOP;
```

Dynamically-sized objects declared in loops 3.3.3.10

Dynamically-sized objects declared in the body of a loop inhibit vectorization.

Example:

```
For I in 501 .. 3000 loop
  Declare
  S : Array(1 .. I) of Integer;   —This declaration inhibits vectoriztion

  begin
    S(I - 500) := S(I) + 2;
  end;
end loop;
```

Loops with dependencies 3.3.3.11

Some loops contain multiple dependencies that cannot be removed by statement reordering. Consider the following case:

```
A : array (INTEGER range 1 .. 101) of integer;
B : array (INTEGER range 1 .. 101) of integer;
C : array (INTEGER range 1 .. 101) of integer;
D : array (INTEGER range 1 .. 101) of integer;

. . .

FOR I IN 1..100 LOOP
  C(I+1) := C(I) * A(I) + D(I);
  D(I+1) := C(I+1) * B(I) + D(I);
END LOOP;
```

There are dependencies on both object C and object D that cannot be removed by the compiler or by simple manual reordering.

Conditional dependencies 3.3.3.12

Some loops have dependencies that the compiler cannot determine at compile time because the subscripts are based on variables that are not known until execution time. These loops do not vectorize. Two examples are the following:

```
A : array (INTEGER range 1 .. 101) of integer;
K : Integer;

. . .

For I in 20 .. 80 loop
  A(I) := A(I - K);
End Loop;
```

If $K \leq 0$, or $K \geq 64$ (the Cray Research systems' vector length), there is no dependency, but if $0 < K < 64$, this is a true dependency. Because of the runtime checks required, however, Cray Ada does not vectorize either case.

Consider the following example:

```
A : array (INTEGER range 1 .. 101, 1 .. 101) of integer;

. . .

For I in 1 .. 100 loop
  For J in 1 .. I - 1 loop
    A(I,J) := A(J,I);   —This loop does not vectorize
  end loop;
end loop;
```

This loop will not vectorize, because the compiler cannot determine that the subscripts do not overlap.

Array with noncontiguous index ranges 3.3.3.13

Arrays declared with noncontiguous index ranges do not vectorize. An example of using enumeration types follows:

```
TYPE Error_Level IS (Comment, Warning, Fatal, Abort);
FOR Error_Level USE (Comment => 2, Warning => 6, Fatal => 7, Abort =>
11);
TYPE Error_Levels IS ARRAY(Error_Level) OF Integer;
Levels : Error_Levels;

. . .

FOR Level IN Error_Level LOOP
  Levels(Level) := Levels(Level) + 1;   —This loop will not vectorize
END LOOP;
```

Scalar recurrence 3.3.3.14

A *scalar recurrence* is an instance in which a scalar object is used in a loop, then later changed in the loop. The loop cannot vectorize, because the value from the previous iteration is necessary to complete the next iteration. An example of this is the following:

```
A : array (INTEGER range 1 .. 100) of integer;
C : Integer;
D : Integer;

. . .

For I in 1 .. 100 loop
 A(I):= A(I) + D;
 D:= A(I)*3;
End Loop;
```

Slice operations
3.3.3.15

Slice operations inhibit vectorization. The following is an example of a nonvectorizable loop with slices:

```
A : array (INTEGER range 1 .. 200) of float;
B : array (INTEGER range 1 .. 400) of float;

...
FOR I IN 1..10 LOOP
  FOR J IN 5..60 LOOP

    A(10 * I ..10 * I + 20) := B(J.. J + 20);
  END FOR;
END FOR;
```

Vectorization messages
3.3.4

The vectorization process may result in informative messages that indicate the following:

- Loops that vectorize
- Loops that do not vectorize

  The reasons that loops do not vectorize are listed with the error messages. These messages may be used to assist in making changes in order to make loops vectorizable. The terminology used in these messages is explained in *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

- Performance information

  Information given in these messages is intended to assist in enhancing the performance of vectorized code.

A failure in the vectorization process is signalled with an internal error. When this occurs, vectorization of the current loop is halted and the loop is compiled without vectorization.

For a detailed explanation of possible error messages, see *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

Performance considerations 3.3.5

Users of Cray Ada 2.0 may enhance the vectorizability of a program. Several general rules should be followed:

- Define numeric types with as tightly constrained a range as possible. In the following, the first would be used rather than the second.

  ```
  SUBTYPE R1 IS Positive RANGE 1..100;
  V1 : R1;

  V1 : Integer;
  ```

  The range of values that an Ada object may take is used during vectorization to determine whether dependencies may occur. This in turn determines not only whether vectorization is possible but also how efficient vectorized code may be.

- Minimize the complexity of `IF` and `CASE` statements

  In general, the more complex the statements, the slower, the vectorized code will be. Complexity consists of the number of `ELSIF` or `WHEN` statements, the complexity of Boolean expressions for `IF` and `ELSIF` statements, and ranges of values and numbers of alternatives for a `WHEN` statement.

- Larger loops provide more opportunities for optimization.

  The quality of vectorized code may be enhanced by the inclusion of more code in a vectorizable loop. The chaining abilities of the CRAY Y-MP and CRAY X-MP systems may be put to better use. An example of implementing the matrix expressions with calls to matrix arithmetic routines is the following:

  ```
  A := B + C * D
  ```

The previous example is evaluated as the following:

```
T1 := Times(C,D)
A := Add(B,T1)
```

This is less efficient than performing all of the operations in a single loop. Although the matrix routine approach appears at first to be a good technique, it is in actuality much less efficient than a single loop approach.

The following is much more efficient:

```
FOR I IN A'First..A'Last LOOP
  A(I) := B(I) + C(I) * D(I);
END LOOP;
```

User-selectable optimization features 3.4

The following subsections describe user-selectable optimization features.

Guidelines for using in-line expansion 3.4.1

You can control inlining in the following four ways:

- Insert pragma `INLINE` designations in the source code, indicating the subprograms to be inlined; the optimizer inlines them.
- Specify the subprograms to be inlined; they are inlined as through pragma `INLINE` had been inserted in the source.
- Allow the automatic inlining of all subprograms called from only one place in a program, including those you do not specify.
- Exempt selected subprograms from automatic inlining; you still get the benefits of the automatic feature.

Inlining is controlled by the `-O` option on the `ada` and `aopt` command lines. The following subsections explain the use, requirements, and effects of inlining.

Using pragma `INLINE` 3.4.2

Cray Ada supports the inlining of calls to subprograms that users identify through `INLINE` pragmas. As specified by the LRM, the pragma must be placed in the same declarative region as the declaration of the subprogram to be inlined and must follow the subprogram declaration. In the following example, the directive to in-line function `Drag_Coeff` in the package declaration for package `Drag_Calc` is placed after the declaration of the function:

```
package Drag_Calc is

 type Plane_Type is (B707, B727, B737, B747, B757, B767);

 function Drag_Coeff (Plane: Plane_Type) return Float;
 pragma INLINE (Drag_Coeff);

end Drag_Calc;
```

If you do not specify `-O on` (at a minimum) as an option on the `ada` or `aopt` command line (that is, `-O off` cannot be specified), pragma `INLINE` is ignored.

Using automatic inlining 3.4.3

Cray Ada also supports *automatic inlining*, in which subprograms that are called from only one place and that are not visible outside the compilation unit or the collection of units being optimized are inlined automatically. Such subprograms are commonly encountered in two instances:

- A programmer has separated out a functional block of code as a subprogram to keep the size of the caller's source down to a manageable level. The optimizer helps to eliminate the penalty for this style of structured design. Elimination of the call overhead by inlining is especially beneficial when the subprogram will be called inside a loop that is repeated many times.

- The compiler has inserted one-shot calls to compiler-generated local subprograms to simplify the implementation of various language features, such as tasking. When inlining is enabled, the compiler inlines these subprograms automatically.

Using transitive inlining 3.4.4

The inlining of subprograms is transitive. For example, if inlined subprogram A is called by inlined subprogram B, and B is called by subprogram C, optimization will result in A being inlined in B, and the result being inlined in C.

Using transitive inlining 3.4.4

The inlining of subprograms is transitive. For example, if inlined subprogram A is called by inlined subprogram B, and B is called by subprogram C, optimization will result in A being inlined in B, and the result being inlined in C.

The one exception to this rule is that a subprogram is not inlined automatically into a subprogram that is itself marked for inlining, using pragma `INLINE`. Automatic inlining is inhibited to ensure that you have full control over the inlining process. This feature prevents any significant unexpected and undesired size overhead introduced by the automatic inlining of a called subprogram. Any subprogram that is to be inlined into another inline subprogram must be marked explicitly with an `INLINE` pragma.

Performance trade-offs 3.4.5

In-line expansion is one type of optimization for which space/time trade-off is an issue. A subprogram that you have marked for in-line expansion and that is called from more than one place can potentially cause object code to be larger after optimization than before if the inlined subprogram has significant size. Usually, subprograms identified for inlining should be small enough that expansion takes little, if any more space than the call it replaces. In any case, in-line subprogram designations are honored regardless of code space costs; therefore, it is up to you to evaluate potential trade-offs.

Requirements for inlining 3.4.6

For a subprogram to be inlined in a unit that calls it, the following conditions must be met:

1. The subprogram must be designated in an `INLINE` pragma or be subject to automatic inlining through the `-O` option of the `ada` or `aopt` command.

2. The unit containing the subprogram to be inlined must be optimized through, as a minimum, the `-O on` option (that is, `-O off` cannot be specified).

3. A unit that calls the inlined subprogram must be optimized.

   Conditions 2 and 3 indicate that both the called subprogram and the code that calls it must be optimized if inlining is to occur.

4. A unit that contains the body of an inlined subprogram must be compiled before the compilation of any units that call the inlined subprogram.

5. Full intermediate code forms (saved with the `-k` option of the ada command) of the unit containing the subprogram to be inlined must be present in the Ada library.

   Inlining fundamentally consists of inserting the code for the inlined subprogram into the calling subprogram. Thus, the code for the inlined subprogram must already exist if this insertion is to take place.

If any of these conditions are not met, inlining will not occur and a normal call to a noninlined copy of the subprogram will occur.

Unit dependencies created by inlining 3.4.7

Because of the nature of the Ada language, inlining may create new unit dependencies. Programmers must anticipate the consequences of inlining certain subprograms within a given configuration.

Callable bodies for visible in-line subprograms 3.4.7.1

Because of the possibility that a caller has been compiled before the compilation and optimization of an in-line body, the compiler always generates a callable body for a pragma `INLINE` subprogram that is externally visible. Generation of a callable body can be avoided by declaring the subprogram where it is not externally visible (for example, in the body of a package) or by designating the unit to be a hidden unit of a collection that includes all of the subprogram's callers.

In-line body dependencies 3.4.7.2

When a subprogram call is expanded inline, a dependency is created between the unit body in which the expansion occurs and the unit containing the inlined body. Recompilation of the in-line body causes the unit in which the expansion occurred to become obsolete. Unless the unit containing the in-line expansion is subsequently recompiled, an inconsistency will be detected when an attempt is made to rebind the main program.

In the following example, assume that main program procedure `Glide_Ratio` calls inlined function `Drag_Coeff` in package `Drag_Calc` and that both procedure `Glide_Ratio` and package `Drag_Calc` have been optimized.

```
package Drag_Calc is
type Plane_Type is (B707, B727, B737, B747, B757, B767);
function Drag_Coeff (Plane: Plane_Type) return float;
pragma INLINE (Drag_Coeff);
end Drag_Calc;
```

If the body of package Drag_Calc is recompiled, unit Glide_Ratio will be rendered obsolete because the actual body object code of function Drag_Coeff has been placed into the object code of Glide_Ratio.

When the body of Drag_Calc is recompiled, Glide_Ratio must be recompiled so that the new (potentially modified) code of Drag_Coeff is included.

Without optimization, a recompilation of the body of package Drag_Calc would not require a recompilation of Glide_Ratio.

To anticipate the consequences of inlining, you may use library command arel to obtain a dependency report involving the relevant units. You may also use arec to check library consistency at any time and to produce a list of units that require recompilation. (See *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014, for more information on these library utilities.)

Pragma VECTORIZE_LOOP 3.4.8

Pragma VECTORIZE_LOOP is an implementation-dependent pragma that gives the Cray Research system vectorizer greater latitude in vectorizing loops. It allows the vectorizer to vectorize some loops for which there is insufficient information to determine vectorizability.

Optimization Utilities and Strategies [4]

Optimization Utilities and Strategies [4]

Using the optimization facilities of Cray Ada, you can significantly increase the execution performance of your Ada codes. You can often gain additional performance by locating time-consuming code and modifying it. This subsection describes various tools available on Cray Research systems that let you fine tune programs for maximum performance.

`apro`
4.1

The Ada profiler `apro` is a tool that helps you gather information on subprogram calls and timings. The functionality of this tool and examples of its' usage are described in *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

Optimization information
4.2

The Cray Ada compiler provides information on specific loops that have and have not vectorized and inlined. The Ada compiler also provides details on how to modify code to support vectorization. Additional information related to this feature can be found in *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

`prof` and `profview`
4.3

The UNICOS `prof` utility indicates the amount of time spent in various segments of code within routines. Usually, it is used in conjunction with `profview`, an interactive tool for displaying the information collected by `prof`. These tools can help you in determining areas on which to focus optimization efforts, especially when you are using mixed languages, because `apro`, the Ada profiling tool, provides no support for foreign languages.

Sample output of an alphabetical list of modules using the a option of profview is shown in the following.

```
Module Name   Hit Count   PCT   ACCUM %
------------- --------- ------ ------
$expand               1   0.08   0.08
creat                 1   0.08   0.17
sbreak                2   0.17   0.34
test_$2            1187  99.66 100.00 ********
```

The information provided by prof and apro does not always map one-to-one. For instance, prof provides information on Ada routines by using internal naming conventions and UNICOS library calls made by the Ada run time. apro, on the other hand, reports at the Ada unit level but does not provide information on UNICOS routines used in the Ada run time. apro does, however, provide an analysis of program calling structure not available with prof; therefore, both tools can be useful in the tuning process.

procstat and procrpt 4.4

The procstat and procrpt routines monitor the execution of processes and generate statistics about I/O process and memory activities. See the *UNICOS Performance Utilities Reference Manual*, publication SR-2040, for more information.

To use procstat, you could use the following command:

```
procstat -R raw.data async17
procrpt raw.data >&! proc.report
```

hpm

4.5

The hpm tool provides information about hardware performance during execution of a program. This tool is available only on CRAY Y-MP and CRAY X-MP systems. No special requirements are necessary to use this routine. For example, using the following command produces output from hpm:

```
hpm mflops
```

The following example represents output from using the hpm mflops command.

```
Group 1:  CPU seconds   :   169.08103    CP executing:  28189568830

  Hold issue condition           % of all CPs       actual # of CPs
Waiting on semaphores            :   0.00                     44858
Waiting on shared registers      :   0.00                         0
Waiting on A-registers/funct.units : 19.69               5551338711
Waiting on S-registers/funct.units : 44.88              12650916996
Waiting on V-registers           : 0.05                    12686456
Waiting on vector functional units :  0.00                      432
Waiting on scalar memory references:  0.02                  4612724
Waiting on block memory references :  0.26                 72404149
```

See *UNICOS Performance Utilities Reference Manual*, publication SR–2040, for additional information on the usage of hpm.

Ada Program Runtime Model [5]

Page

Page

Figures

Tables

Ada Program Runtime Model [5]

The Ada runtime model encompasses all aspects of the structure necessary for Ada to communicate with the UNICOS operating system and user environment. This model includes information on the following areas:

- Ada runtime support packages
- System and library calls
- Internal representation of data
- Storage and task management
- Exception handling
- Cray Ada calling sequence

Overview of the Ada Execution Environment
5.1

The Cray Ada Environment's full Ada runtime environment is called the *Ada Execution Environment* (AEE). It includes support for built-in language facilities (such as tasking and memory allocation) and support for the predefined packages (such as `Text_IO` and `Calendar`) that must be explicitly included by use of `with` to become accessible to a user program.

One key set of target-independent runtime components is known collectively as the *Runtime Support Packages* (RSP). Conceptually, each Ada program implicitly imports the RSP packages it needs, although the RSP specifications are not made available to the user. When runtime support is required, the compiler inserts calls to the RSP into the core of a user program. The RSP supports some of the more complicated aspects of Ada semantics, such as tasking and dynamic memory management. The RSP is implemented entirely in Ada.

The Target-dependent RSP (TDRSP) provides the foundation for the RSP. The TDRSP includes several packages with target-independent specifications and target-dependent bodies that implement the fundamental low-level primitives on which the RSP depends. For instance, one of the packages (`TD_Machine_State_Manager`) provides the machine-level context switch operation that occurs as part of an overall Ada task switch.

Contents of the runtime sublibrary 5.2

The runtime sublibrary contains the Ada packages required to support program compilation, linking, and execution. It is sometimes referred to (loosely) as the *runtime library* or the *standard library*. As previously mentioned, the standard predefined Ada sublibrary must be specified as part of the library for each compilation and linking. The I/O operations provided by the runtime library are performed synchronously, with program execution suspended until the I/O operation is complete.

The following are the units in the runtime library:

- Ada predefined units. The standard library consists of the following Ada predefined units, which you must not redefine:

  ```
  Calendar
  Direct_IO
  IO_Exceptions
  Sequential_IO
  Standard
  System
  Text_IO
  Unchecked_Conversion
  Unchecked_Deallocation
  ```

  If you redefine these units, you may get unpredictable program results.

- Preinstantiated I/O packages. Besides the Ada predefined packages, the standard library provides the following preinstantiated versions of packages in standard package `Text_IO`:

  ```
  Float_Text_IO
  Integer_Text_IO
  ```

  The specifications of these packages, and information on their use, are presented in "Use preinstantiated generic units," page 6.

- **Low-level support packages.** Besides Ada predefined and preinstantiated I/O packages, the runtime library consists of the support packages listed in Table 5.

Caution

These packages are for implementation purposes only and are not supported for direct user access. If you attempt to call them from a user program, the program may abort or take an otherwise undefined action.

Table 5. Runtime library support packages

Support type	Package names	Support type	Package names
Target-independent tasking support	AR_Abort AR_Activation AR_Attributes AR_Debugger_Support AR_Delay AR_Exception AR_Kernel AR_Rendezvous AR_State AR_TCB_Operations AR_Termination	**Target-independent runtime support**	RSP_Enumeration RSP_Fixed RSP_H1_Open_Heap RSP_H3_Collection RSP_Discrete_Attributes RSP_Real_Attributes RSP_Composite RSP_Vector Telesoft_Integer_Types
I/O support	File_IO Text_IO_Definition UIA URA	**Target-dependent tasking support**	TD_Delay_Manager TD_Exception_Manager TD_Interrupt_Manager TD_Machine_State_Manager TD_Memory_Manager TD_Task_Termination TD_Tasking_Parameters
Target-dependent runtime support	TD_Address_Manager TD_Memory_Manager CGS CGS_Debugger_Support CGS_Exc_Display CGS_Exception_Manager ENV Vmmr_Lib		

- Cray_Lib. The Cray_Lib package contains several routines that may not be called in addition to the user callable routines. Routines that are not for use by other than the Cray_Lib package itself begin with Q8. See "Library Interfaces," page 119, for a description of Cray_Lib.
- System_Info. The System_Info package contains the compiler version number in both number and string format. See "Library Interfaces," page 119, for a description of System_Info.
- UNICOS_Signal_Support. The UNICOS_Signal_Support package contains user callable routines. See "Library Interfaces," page 119, for a description of UNICOS_Signal_Support.

Reserved global names
5.3

The Cray Ada compiler runtime contains a collection of modules that forms a direct interface with UNICOS. These modules are written in a combination of C and CAL. Routines written in languages other than Ada (CAL, C, Fortran, Pascal, and so on) should avoid redefining the global names listed in "UNICOS system calls and C library routines," and "UNICOS global data items," page 77, because they are used by the Cray Ada compiler runtime. Redefinition of these names causes link-time errors and incorrect program execution. The Ada compiler translates package and procedure names into a form that does not conflict with most user-specified names.

UNICOS system calls and C library routines
5.3.1

The following routines are called directly by the Cray Ada runtime and package STANDARD. You can find complete descriptions of these routines for UNICOS 6.0 in *Volume 4: UNICOS System Calls Reference Manual*, publication SR–2012, *Volume 2: UNICOS C Library Reference Manual*, publication SR–2080.

access(2)
close(2)
exit(2)
fcntl(2)
free(3C)
ioctl(2)
localtime(3C)
lseek(2)
malloc(3C)
memcmp(3C)
sleep(3C)
sprintf(3C)
stat(2)
strcpy(3C)
strlen(3C)

getcwd(3C)
getenv(3C)
getpid(2)
getpwnam(3C)
gets(3C)
getwd(3C)
memcpy(3C)
memset(3C)
open(2)
read(2)
rename(3C)
signal(2)
strncpy(3C)
target(2)
time(2)
unlink(2)
write(2)

$STKCR% and $STKOFEN (as in /libc/gen/stackal.s)§
$STKDE% and $STKUFEX (as in /libc/gen/stackde.s)§
$STKOFEN and $STKRETN (as in libc/gen/csus)§§

UNICOS global data items 5.3.2

The Cray Ada Environment uses the following global data items (as defined in the ctime(3C) entry in the *UNICOS C Library Reference Manual*, publication SR–2080). In software, they are defined in the UNICOS libc.a library and are referenced by the UNICOS runtime:

- daylight
- timezone

The Cray Ada Environment uses the following global data items (as defined in the exec(2) and intro(2) entries in *Volume 4: UNICOS System Calls Reference Manual*, publication SR–2012). In software, these items are defined in the UNICOS libc.a library and are referenced by the UNICOS run time:

- environ
- errno

The Cray Ada Environment uses the following global data items on CRAY-2 systems with UNICOS 6.0 or previous levels:

- @argc
- @argv

§ Applies to CRAY Y-MP and CRAY X-MP systems
§§ Applies to CRAY-2 systems

Internal Ada naming conventions
5.4

Some UNICOS systems only accept file names of 14 or fewer characters in length. Because the Ada language allows compilation unit names of much greater length, an internal file-naming convention has been established to identify Ada compilation unit names uniquely within the UNICOS environment. Users of Ada should never be required to reference an Ada unit by these internal names, although the names are referred to when using several of the Ada tools. The following conventions show how these names are generated.

The Ada compiler's middle pass (MP) generates subprograms with names of the following format:

`Mp_L` *source_line_number*`$`*descriptive_name*

For example:

```
Mp_L136$Compatibility_Check_subp
```

If the compilation unit name is longer than 12 characters, the compiler and linker generate unique names for object files by using up to 5 characters of the unit name, appending a processed version of a 7-character time stamp, and then `.o`. Because the time stamp is processed, it does not appear to be a time stamp, but rather a random character string.

The following gives an example:

Ada unit name	Output file name
`A_very_long_name`	`A_verBisw9pc.o`
`A_very_much_longer_name`	`A_verBisw0_3.o`

Specifications start with a capital letter. Bodies use the same time stamp as the specification. If a specification is recompiled, the time stamp will change.

Data representation
5.5

This subsection describes the representation of various types of data in the Cray Ada Environment. It provides summary information about the internal representation of various data types, and it gives examples for packed and unpacked types and unconstrained arrays.

The following list summarizes the default representations of data in the Cray Ada Environment.

- Numbers are represented as follows:

Type	Representation
`integer`	Has an effective range of –2e45 to +2e45–1, although integer objects occupy full 64-bit words, unless they have a range requiring fewer bits and are contained in a packed array or record.
`fixed-point`	Occupies 64 bits with an effective mantissa size of 46 bits and is word aligned. Fixed-point objects cannot be packed. If the upper bound of a fixed-point type is a power of 2 (not a model number for the type), it is not a representable value of the type.
`float`	Provides for an effective accuracy of 13 decimal digits. All floating-point objects are word aligned, and they cannot be packed. For more information on the specifics of the choice of `Max_Digits`, see LRM 3.5.7 Floating-point Types in the LRM annotations of this manual.

- The following rules govern the packing of numeric types:
  - Floating-point objects, whether alone, in arrays or in records, cannot be packed.
  - Fixed-point values, whether alone, in arrays, or in records, cannot be packed. (Type `DURATION` is represented by 46-bit, fixed-point numbers; see, “LRM 9.6: Delay statements, duration, and time,” page 189.)
  - Integers in arrays or records can be packed.

  “LRM 13.1: Representation clauses,” page 192, summarizes the packing of these data structures and provides information about the memory layout of packed types.

- Objects of the predefined type `character` occupy 64 bits and are word aligned, unless packed within an array or record.
- Objects of type `STRING` are packed 1 character per byte (8 characters per word) and are allocated in the same way as packed arrays of a character.

- Objects of type boolean occupy 64 bits and are word aligned, unless packed within an array or record. A value of false is represented by 0; a value of true is represented by 1.

- An enumeration type is represented as an integer range, with the range depending on the number of elements. The space allocated for an object is 64 bits, and the object is word aligned, unless packed within an array or record. The first element of an enumerated type is represented by the value 0, unless superseded by an enumeration representation clause.

- An access type is implemented as a 64-bit absolute memory address on CRAY Y-MP; CRAY-2 systems and cannot be packed. Objects of access type are word aligned. Access types, having a designated subtype of unconstrained array, point to a descriptor record that includes the bounds of the array, followed by the array itself.

- Components in a record follow the preceding data representations. These components are allocated sequentially as declared in the source code (with gaps between components as required to obey alignment rules). Components of 64 bits or fewer never cross word boundaries.

  If pragma PRESERVE_LAYOUT is used, the first field in the record is assigned the lowest memory address; otherwise, in the absence of a representation clause, it may be arbitrarily reordered. Variant record fields are always aligned on word boundaries. Variant records can be packed, subject to the packing rules governing their elements.

- In packed records, arrays are always aligned on a multiple of the size of their individual components (that is, a string is aligned on an 8-bit boundary, an array of Booleans is aligned on a bit boundary, and so on). Components of packed arrays are bit aligned, but never cross word boundaries. If not in a packed record, arrays are always word aligned. Multidimensional arrays are represented in row-major order; that is, the several collections of objects corresponding to the innermost (or rightmost) dimension follow each other sequentially in memory. This order is the same as that used by Cray C and Cray Pascal, and is the reverse of the order used by the various Cray Fortran compilers.

Pragma PRESERVE_LAYOUT 5.5.1

The Cray Ada compiler reorders record components to minimize gaps within records. Pragma PRESERVE_LAYOUT forces the compiler to maintain the Ada source order of a given record type, thereby, preventing the compiler from performing this record layout optimization. The syntax of this pragma is as follows:

```
pragma preserve_layout (ON => record_type_name)
```

Pragma PRESERVE_LAYOUT must appear before any forcing occurrences of the record type and must be in the same declarative part, package specification, or task specification. This pragma can be applied to a record type that has been packed. If PRESERVE_LAYOUT is applied to a record type that has a record representation clause, the pragma applies only to the components that do not have component clauses. These components appear in Ada source code order after the components with component clauses.

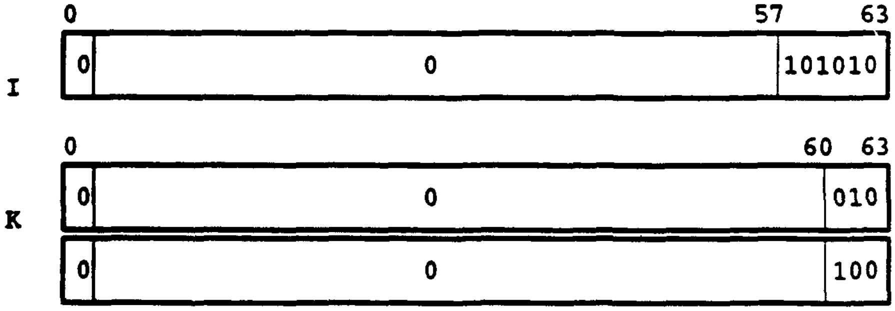
Internal representation of unpacked types 5.5.2

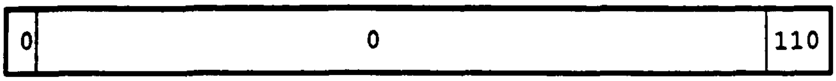
The following subsections show the memory layouts in Cray Research systems for sample objects of Ada's unpacked data types.

Unpacked unsigned integer types 5.5.2.1

The following code segment declares and sets simple unsigned integers and an array of unsigned integers. The resulting objects, when unpacked, are represented in memory as shown in the memory maps following the code segment.

```
subtype small_int is integer range 1 .. 10;
type Short is array (1 .. 3) of small_int;
I : integer := 42;
K : Short := (2,4,6);
```

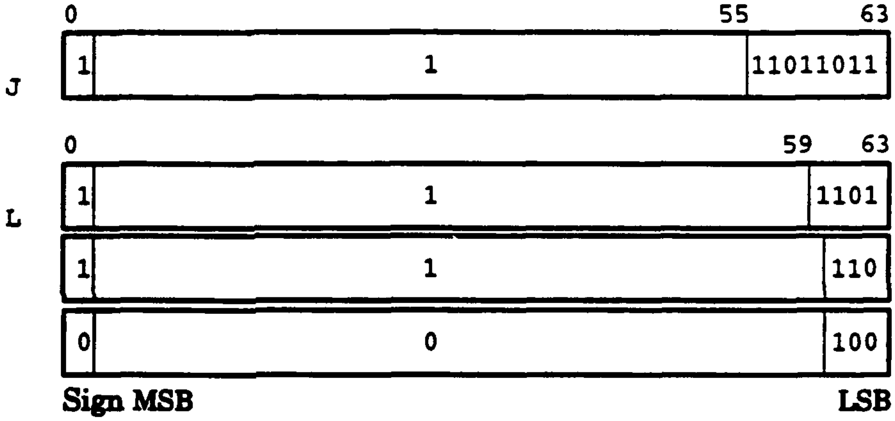




Unpacked signed integer types 5.5.2.2

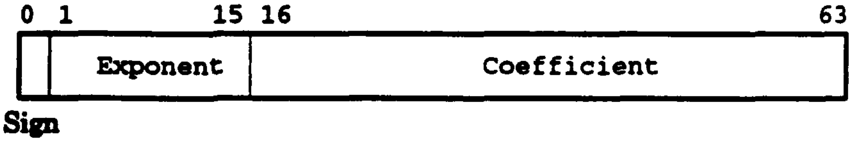
The following code segment declares and sets simple signed integers and an array of signed integers. The resulting objects, when unpacked, are represented in memory as shown in the memory maps following the code segment.

```
subtype small_int_1 is integer range -5 .. 5;
type Short_1 is array (1 .. 3) of small_int_1;
J : integer := -37;
L : Short_1 := (-3,-2,4);
```



Unpacked floating-point types 5.5.2.3

Floating-point values cannot be packed. They are represented in memory as shown in the following memory map:



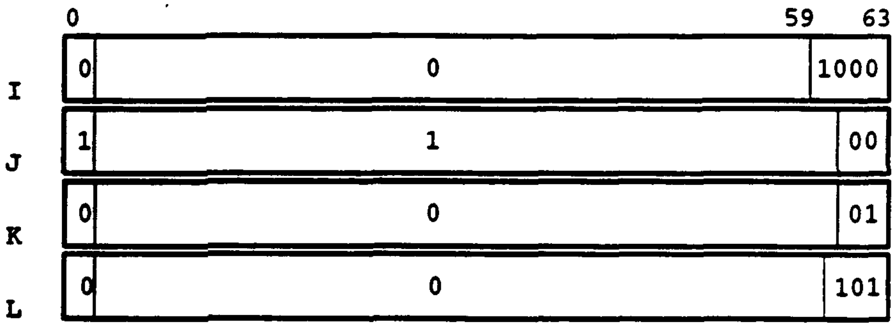
The exponent value has a base value of 40,000 octal added to it; therefore, to get the true exponent, you must subtract this value from the represented exponent.

Unpacked fixed-point types 5.5.2.4

Cray Ada represents fixed-point numbers as the integer number of DELTAS that comprise the number. The DELTA value is computed to be the smallest power of 2 less than or equal to the specified delta. In the first example, the actual DELTA of T1 used by the compiler is T1'SMALL or 1/16 (0.0625). T1'SMALL is the unit value used by the compiler when dealing with fixed-point numbers, and all numbers of the type can be represented as some multiple of this value.

In the following section of code, several fixed-point types are declared and variables of those types are then assigned values. The resulting fixed-point objects are represented in memory as shown in the memory map following the code segment.

```
type T1 IS DELTA 0.1 RANGE -1.0 .. + 1.0;
type T2 IS DELTA 0.2 RANGE -1.0 .. + 1.0;
type T3 IS DELTA 0.5 RANGE -1.0 .. + 1.0;
I : T1 := 0.5;
J : T2 := -0.5;
K : T3 := 0.5;
L : T1 : 0.3;
```



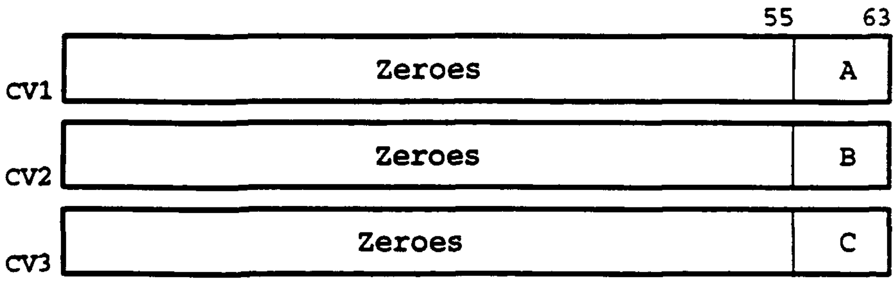
For object I, the number of deltas needed to represent the value is 8; therefore, 8 x .0625 = .5. For object J, the number of deltas needed to represent the value is −4; therefore, −4 x .125 = −0.5. For object K, the number of deltas needed to represent the value is 1; therefore, 1 x .5 = 0.5.

In the preceding examples, the value represented (0.5) was evenly divisible by DELTA. This is not always the case. For object L, the number of deltas needed to represent the value is 5, and 5 x 0.0625 = .3125, which is rounded to the nearest fixed-point number, 0.3.

Unpacked character types 5.5.2.5

The following code segment creates three character variables, assigning them values. Those character objects, when unpacked, are represented in memory as shown in the memory maps immediately following the code segment.

```
char_val_1 :character := 'A'; --"CV1" memory map
char_val_2 :character := 'B'; --"CV2" memory map
char_val_3 :character := 'C'; --"CV3" memory map
```



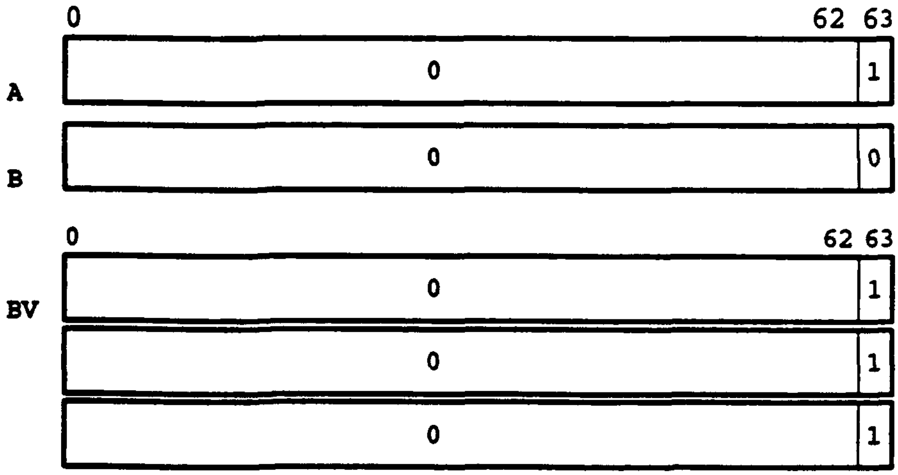
Unpacked string types 5.5.2.6

String types are packed by default in Cray Ada. See "Packed string types," page 87, for details on how packed strings are stored in memory.

Unpacked Boolean types 5.5.2.7

The following code segment sets two Boolean variables and then declares and sets a Boolean array. Those Boolean objects, when unpacked, are represented in memory as shown in the memory maps following the code segment.

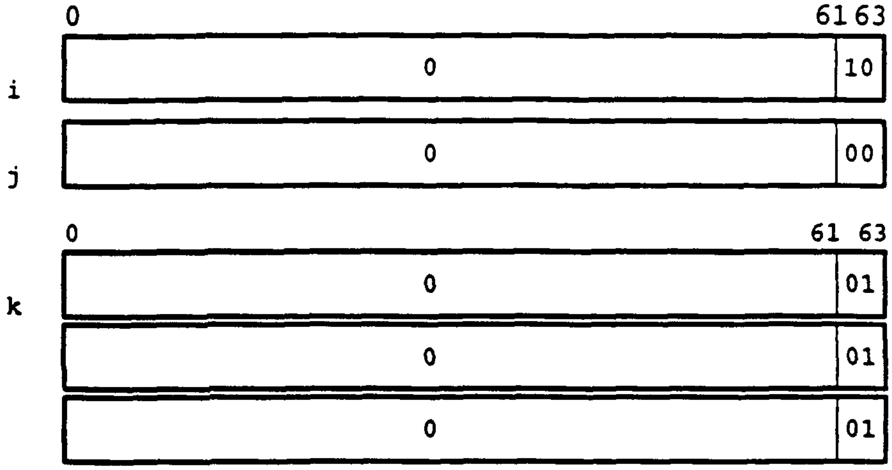
```
A: Boolean := True;
B: Boolean := False;
type Bool_Array is array (1 .. 3) of Boolean;
Bool_Val : Bool_Array := (True, True, True);
--"BV" indicates the boolean array memory map
```



Unpacked enumeration types 5.5.2.8

The following code segment declares two objects of type `enumeration` and then sets three variables. The resulting enumeration-type objects, when unpacked, are represented in memory as shown in the memory maps following the code segment.

```
type Color is (Red,Green,Blue);
type enum_array is array (1 .. 3) of color;
i : color := blue;
j : color := red;
k : enum_array := (green,green,green);
```

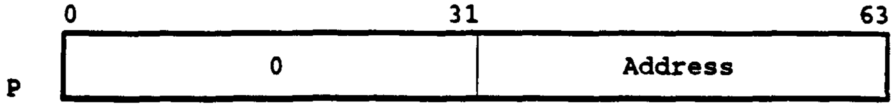


Unpacked access types 5.5.2.9

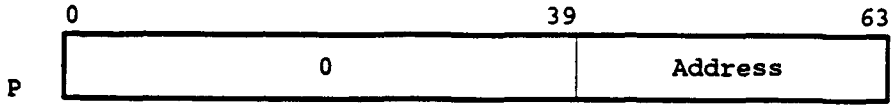
The following code segment defines an `access` type to an array of integers. The resulting `access` type object, when unpacked, is represented in memory as shown in the memory maps following the code segment.

```
type cell is array (1 .. 5) of integer;
type Link is access Cell;
P : Link := new Cell;
```

On CRAY Y-MP and CRAY-2 systems, the object `P` is represented as follows:



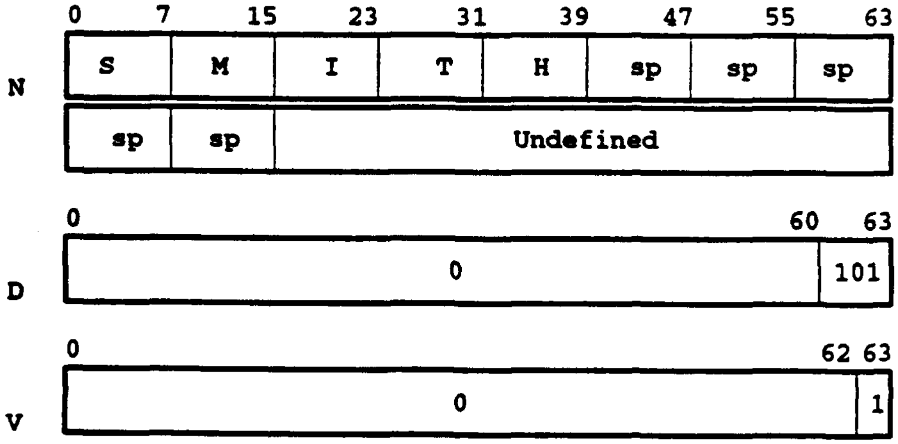
On CRAY X-MP systems, the object P is represented as follows:



Unpacked record types 5.5.2.10

The following code segment declares a record type and sets the fields within the record. The resulting record-type object, when unpacked, is represented in memory as shown in the memory maps following the code segment.

```
procedure Record_file is
type Data is
   record
     Name : String ( 1 .. 10); --"N" in memory map
     Date : Integer;           --"D" in memory map
     Vote : Boolean;           --"V" in memory map
   end Record;
PRAGMA PRESERVE_LAYOUT (ON => Data);
Person : Data := ("SMITH     ", 5, true);
```



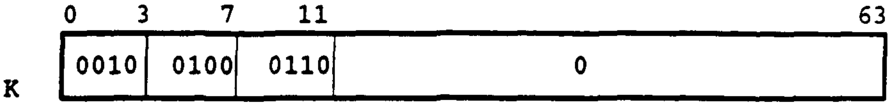
Internal representation of packed types 5.5.3

The following subsections show the memory layouts within Cray Research systems for sample objects of Adas' packed data types.

Packed unsigned integer types 5.5.3.1

The following code segment declares integer types and arrays of integer types and then assigns values to variables of those types. Those variables, when packed, are represented in memory as shown in the memory maps following the code.

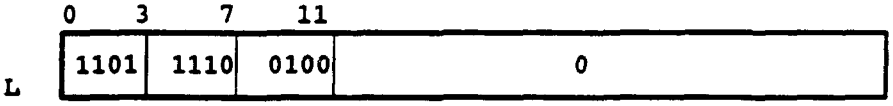
```
subtype small_int is integer range 1 ..10;
type Short is array (1 .. 3) of small_int;
Pragma Pack (Short);
K : Short := (2,4,6);
```



Packed signed integer types 5.5.3.2

The following code segment declares signed integer types and arrays of signed integer types and then assigns values to variables of those types. Those variables, when packed, are represented in memory as shown in the memory maps following the code.

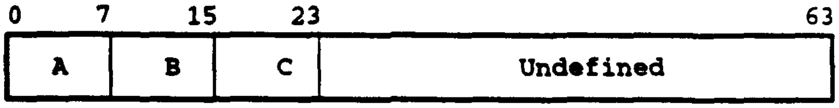
```
subtype small_int_1 is integer range -5 .. 5;
type Short_1 is array (1 .. 3) of small_int_1;
Pragma Pack (Short_1);
L : Short_1 := (-3,-2,4);
```



Packed character types 5.5.3.3

The following code segment declares an array of characters and then assigns it a value. That array object (packed) is represented in memory as shown in the memory map immediately following the code segment.

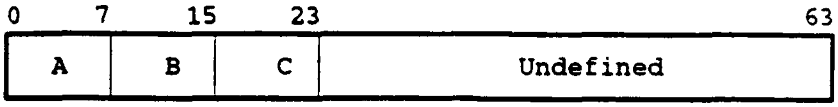
```
type Char_Array is array (1 .. 3) of Character;
PRAGMA PACK(Char_Array);
Char_Val : Char_Array := ('A', 'B', 'C');
```



Packed string types 5.5.3.4

Strings in Cray Ada are always packed by default and do not require a `pragma PACK` statement. The following code segment declares a string subtype and then assigns a value to a string object of that subtype. That string object is represented in memory as shown in the memory map following the code segment.

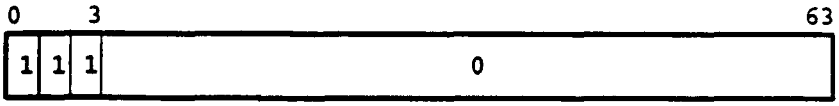
```
subtype String_type is string (1 .. 3);
s_t_String : String_type := ("ABC");
```



Packed Boolean types 5.5.3.5

The following code declares a Boolean array and assigns values to it. The resulting object, when packed, is represented in memory as indicated by the memory map following the code.

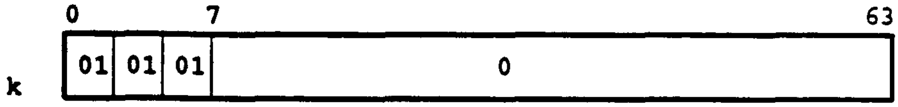
```
type Bool_Array is array (1 .. 3) of Boolean;
Pragma Pack (Bool_Array);
Bool_Val : Bool_Array := (True, True, True);
```



Packed enumeration types 5.5.3.6

The following code segment declares an enumeration type and assigns it a value. The result, when packed, is represented in memory as shown in the memory map following the code segment.

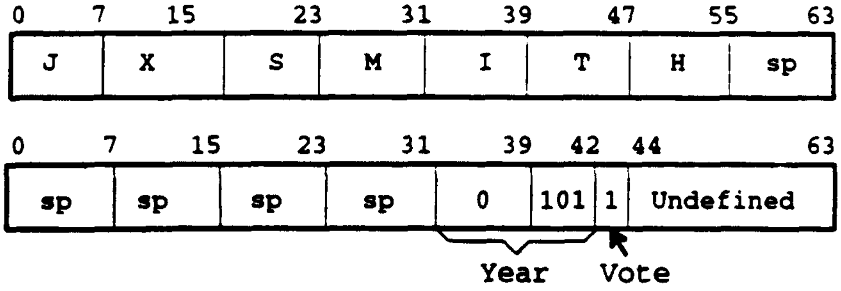
```
type Color is (Red,Green,Blue);
type enum_array is array (1 .. 3) of color;
Pragma Pack (enum_array);
k : enum_array := (green,green,green);
```



Packed record types 5.5.3.7

The following code segment declares a record-type object, assigning it values. The resulting record, when packed, is represented in memory as shown in the memory map following the code segment.

```
subtype Small_int is integer range 0 .. 2000;
type Data is
   record
     Init_1 : Character;
     Init_2 : Character;
     Name : String ( 1 .. 10);
     Year : Small_int;
     Vote : Boolean;
   end record;
PRAGMA Pack(Data);
PRAGMA PRESERVE_LAYOUT (ON => data);
Person : Data := ('J', 'X', "SMITH     ", 5, true);
```



Internal representation of unconstrained arrays 5.5.4

Unconstrained arrays have a descriptor block associated with them. The following subsection discusses the use of unconstrained arrays and the descriptor block information associated with them.

Memory layout of unconstrained arrays 5.5.5

When using unconventional means (such as `'address` or an access type generated by an unchecked conversion) to reference an unconstrained array, the address pointed to is a descriptor block containing information about the unconstrained array. The size of this descriptor block depends on the data type and dimensionality of the array. Because the descriptor block size differs with each situation, when using this sort of structure users must be careful to address the correct structure, and they may need to determine the number of words being allocated for the descriptor block of the specific unconstrained array. The `'size` attribute provides the actual size of the array, not including the descriptor block. Access to the descriptor block is not needed by application programmers, so care must be taken to address the array correctly. The following example shows one way a user can obtain the various addresses to the array:

```
with System;
 procedure vect_test_1b is
    type vect_array is array (integer range <>) of integer;
    type point is access vect_array;
    subtype vect is vect_array (1 .. 5);
    i : point;
    k : vect;
    L, M, N : system.address;
 begin
    i := new vect'(others=>5);
    i(1) := 1;
    i(2) := 2;
    i(5) := 15;
    i(3) := 7;
    L := i'address;               — address of access variable
    M := i.all'address;           — address of descriptor block
    N := i(i'first)'address;      — address of first element
 end vect_test;
```

The following shows a sample memory layout for an unconstrained array:

```
    00120642:   0000000000000000134657

       ...               ...

    00134657:   0000000000000000000005
    00134660:   0000000000000000000001
    00134661:   0000000000000000000005
    00134662:   0000000000000000000001
    00134663:   0000000000000000000001
    00134664:   0000000000000000000002
    00134665:   0000000000000000000007
    00134666:   0000000000000000000005
    00134667:   0000000000000000000017
    00134670:   0000000000000000000000
    00134671:   0000000000000000000000
```

In the preceding example, object L contains the address to access type i. In the preceding memory diagram, this value is stored at memory location 120642. The access type points to the descriptor block of the array. Object M contains the address to the descriptor block for the unconstrained array. This descriptor block begins at address 134657 in the example. Object N provides the address to the first element of the array. In the example, the array begins at address 134663 and continues through address 134667. The value in object N is most likely the address you may need.

Caution

The sizing of descriptor records may change from release to release and has no bearing the way other vendors may store descriptor information. Using 'address or an access type to reference the descriptor block of an unconstrained array is done at the programmer's risk. CRI does not guarantee consistent sizing of descriptor records.

Storage management
5.6

The main program and all tasks in it share a single heap used for all dynamically allocated storage that the program requires. Stack space for Ada tasks, including the main program task, is allocated on the heap.

Heap allocation and deallocation requests are ultimately satisfied by calls to routines in the RSP storage management module. These routines manage a heap composed of one or more contiguous areas of memory called *pools*. If the available pools cannot satisfy an allocation request, a routine is called to allocate another pool area, which will be noncontiguous. `access` type collections are chained together and released when the access type declaration scope is exited.

If a constrained subtype of an unconstrained array is freed using unchecked deallocation, the available memory is returned to the program heap. This memory is now available for reuse by the program. Therefore, if a new array is defined dynamically, that space may be reused by the memory manager to represent the structure, provided there is sufficient memory in that heap pool.

Task management
5.7

The Cray Ada runtime manages tasks through a variety of steps. Tasks are first elaborated, then activated, then executed, with a number of conditions possible for each step. This subsection discusses the steps in task management, how tasks are scheduled, and how you can synchronize them and delay them. Tasks may not register for hardware interrupts in this implementation.

Task elaboration 5.7.1

When a task is elaborated, a stack is allocated on the heap for the task to use. You can specify a task's initial stack size by using a `length` clause. If you do not use a `length` clause, the initial stack size is the default of 4000 words. It may also be specified by using the `-Y` switch when binding.

If the initial stack size is insufficient, a task's stack automatically grows dynamically. A `STORAGE_ERROR` exception occurs only when there is no more space in the heap for additional stack segments. Expansions to the stack are made by calls to `$STKOFEN`, taking the system default for the size of the increment requested. The default increment size is site-configurable, so check with your system support staff if you need to know what the value is for your system. A task's stack is deallocated when the task terminates.

A Task Control Block (TCB) is allocated on the heap for each task and is deallocated only when the master scope for the task is terminated.

Task delay 5.7.2

The `delay` statement is implemented using the system clock. If an Ada program is rolled out of execution by the operating system immediately after processing of a `delay` statement of *x* seconds and is subsequently rolled back in after being suspended for *y* seconds, the delay is considered to have expired if *y* is greater than *x*. That is, the time for which a job is suspended counts against `delay` times in that job, the clock keeps running even when the job is suspended.

Note

Because a real-time clock is used to monitor time, and the Ada program is not the only process on the system, the execution order of Ada tasks is affected by the priority of the Ada program and the system workload.

Task rendezvous 5.7.3

Tasks that call entries in other tasks are placed on an entry queue, specific to the entry point they call, in first-come, first-served basis. In addition to being placed on a priority-based execution queue, a task making a timed entry call is also placed in the delay queue based on the specified delay period.

The server or caller (whichever comes last) places the server on the ready queue so that the rendezvous can occur. A stack of callers is maintained during the rendezvous to allow for nested accept statements and the proper relinquishing of the appropriate caller at the end of a rendezvous.

Pragma PRIORITY does not affect the selection from pending entry calls in a selective wait statement.

Parameter passing at a rendezvous depends on a parameter block built by the caller at the time of the call. A pointer to the block is passed to the called task when it reaches the rendezvous point.

Task scheduling 5.7.4

This subsection describes the way in which the Ada runtime schedules tasks for execution based on their relative priorities and on the synchronization points and entry calls for which they may wait.

A task is placed on one of 64 different execution queues based on its priority 1 through 64; 64 is the highest priority. You can explicitly set a task's priority using pragma PRIORITY, or accept the automatic default priority of 31. The ordering of tasks of equal priority (within the same execution queue) is indeterminate. The first task in the highest priority queue is activated first and runs until it reaches a synchronization point. All tasks in the highest priority queue are activated and run first. When the highest-priority queue is empty, or all its tasks are blocked, tasks from the next-highest priority queue are activated and run. This process continues until all tasks in all priority queues have terminated.

If an executing task becomes blocked by execution of an Ada `delay` statement, it is placed on the delay queue. If a task is blocked because it is waiting for a synchronization point, it is placed on the bottom of the execution queue from which it was taken. If a task is blocked because it is waiting on an entry call to another task, it is placed on the entry queue for that entry point.

Tasks are placed in the entry queue and the execution queue in first-in, first-out (FIFO) order. Tasks are placed in the delay queue in the order of their specified delays, with the shortest delay at the front of the queue.

When a task reaches a synchronization point or executes a `delay` statement, a task switch occurs if another task of higher priority is ready to run (having completed a wait). A *task switch* is the transfer of the executing task from executing to either termination or a queue, and the consequent transfer of another task from its execution queue to executing (see LRM 9.11(2)).

Note

Because the Ada runtime does not use time slicing when executing tasks, a task can hog the CPU if it is allowed to execute without reaching a delay or a synchronization point, such as a rendezvous.

The Ada tasking model implemented in the Cray Ada Environment does not currently support multiple CPUs. Future releases of the Cray Ada Environment will support the division of Ada tasks among multiple CPUs. Because of the current single-CPU implementation, tasking reduces throughput rather than increases it, due to the tasking overhead.

Exception handling 5.8

Each declared exception is identified by an address that points to a location to which the exception name resides. When an exception is raised, the address of the exception identifier and the address at which the exception occurred are passed in a call to the runtime system. The occurrence address is then used to determine in what scope the exception occurred. After that, the dynamic link is followed until an exception handler that matches the exception name is encountered. Exceptions cause no runtime overhead unless raised. The presence of exception handlers, however, can inhibit certain optimizations. The Cray Ada Environment adheres strictly to the requirements of exception handling within tasks.

By default, the only signal that Cray Ada traps is `SIGFPE` (UNICOS signal 08). This is mapped to the Ada exception `numeric_error`.

A package called `UNICOS_Signal_Support` allows an Ada program to trap most UNICOS signals. When enabled, these signals are mapped to the exception `UNICOS_SIGNAL`. There are no default mappings of UNICOS signals to `UNICOS_Signal`. Any mappings must explicitly be turned on by the `Set_Signal_Mapping` routine.

Unhandled exceptions 5.8.1

When an unhandled exception occurs, a message of a standard format is sent to the standard error, indicating the type of the exception and where it was raised. For the predefined exceptions, additional information is provided about the condition that raised the exception. The general format of these exception messages is as follows:

```
>>> UNHANDLED EXCEPTION <<<

 Exception : exception_type
 Reason    : exception_condition

      raised in unit_name.proc_name at line line_num

      called from unit_name. proc_name at line line_num
      called from unit_name.proc_name at line line_num

        . . .
```

The *exception_type* is the type of exception raised.

The *exception_condition* describes the cause of the exception for predefined exception types.

The *unit_name* is the name of the unit in which the exception was first raised.

The *proc_name* is the Ada name of the innermost scope in which the exception occurred.

The *line_num* is the line number of the source file at which the exception occurred.

The *exception_type* field will be one of the five predefined exception types or a user-defined type. For the five predefined types, this consists of one of the following strings:

```
constraint_error
program_error
storage_error
numeric_error
tasking_error
```

For user-defined exceptions, the *exception_type* field will be of the following form:

exception_name

The *exception_name* is the Ada name of the exception, as declared by the user.

For predefined exceptions, *exception_condition* is the string that indicates the reason for the exception. Table 6 presents the possible exception conditions that may occur for each of the five predefined exceptions. For user-defined exceptions, this item will not be present.

The *unit_name* and *proc_name* identify the scope in which the exception was first raised. The *unit_name* is the Ada name of the unit in which the exception was raised. The *proc_name* is the Ada name of the innermost named scope in which the exception was raised.

The remainder of the message consists of the sequence of subprogram calls that resulted in the call to the scope in which the exception occurred. These calls are presented in reverse order, with the most recent calls appearing first in the list.

Table 6. Exception conditions for predefined exception types

exception_type	*exception_condition*

`constraint_error:`	`Access check`
	`Discriminant check`
	`Index check`
	`Length check`
	`Range check`
	`Range/index check`

`numeric_error:`	`Division by zero`
	`Numeric overflow`

`program_error:`	`Subprogram elaboration`
	`Generic elaboration`
	`Function without return`
	`Task elaboration`

`storage_error:`	`Allocator failure`
	`Stack overflow`
	`Task stack allocation`

`tasking_error:`	`Select statement unopen`
	`Child activation`
	`Task not callable`

Exception reporting 5.8.2

A new function, `System.Report_Error`, has been added to the system package. `System.Report_Error` can be called by the user program from any exception handler. It prints a traceback of the most recently handled exception, including a traceback from the point of the call to `System.Report_Error` itself.

In the following example `proc1` is called, `proc2` is called from `proc1`. An exception is raised in `proc2`, `proc1` handles it and `System.Report_Error` is called and the traceback is printed. The program then continues.

```
with Text_IO;
with System;
procedure treperr is

  My_Error : Exception;

  procedure proc2 is
  begin
    Text_IO.Put_line ( "Raising exception...");
    raise My_Error;
  end proc2;

  procedure procl is
  begin
    proc2;
  exception
    when My_Error =>
    Text_IO.Put_line("Handling exception ...");
    System.Report_Error;
  end procl;

begin
  procl;
  Text_Io.Put_line( "Continuing after handling exception...");
end treperr;

Output
------

Raising exception ...
Handling exception ...

>>> Exception traceback from Report_Error <<<
  Exception: MY_ERROR

    raised in sec/treperr.proc2 at line 10

    called from sec/treperr.procl at line 15
     Report_Error invoked in sec/treperr.procl at line 19

    called from sec/treperr.treperr at line 23

Continuing after handling exception ....
```

Tasking exceptions 5.8.3

The user may also specify that all exceptions raised in a task, handled or not, be displayed. This may be specified through use of the `-X` switch when binding a program.

UNICOS signal support 5.8.4

The user may also specify that all exceptions raised in a task, handled or not, be displayed. Four routines are provided for setting the specific signals to be caught; only those signals specified through this mechanism (except for `SIGFPE`) are caught. All others pass through and are treated in the default manner defined by UNICOS. This lets you catch signals from other languages if you so choose.

When a particular signal is set and caught, you will not trap that signal again, unless you reregister the handler through the signal mapping mechanism. The specifications for this package follow.

```
Package UNICOS_Signal_Support is

  Type Signal is

    (SIGHUP,     -- Hangup
     SIGINT,     -- Interrupt
     SIGQUIT,    -- Quit
     SIGILL,     -- Illegal instruction
     SIGTRAP,    -- Trace trap
     SIGABRT,    -- Hardware error
     SIGERR,     -- Error Exit
     --SIGFPE,   -- Floating-point exception (converted to Numeric_Error)
     --SIGKILL,  -- Kill (Cannot be caught or ignored)
     SIGPRE,     -- Program range error
     SIGORE,     -- Operand range error
     SIGSYS,     -- Bad argument to systemc call
     SIGPIPE,    -- Write on a pipe with no one to read it
     SIGALRM,    -- Alarm clock
     SIGTERM,    -- Software termination from kill
     SIGUSR1,    -- User defined signal 1
     SIGUSR2,    -- User defined signal 2
     SIGCLD,     -- Death of a child process    (This is not supported in Ada)
     SIGPWR,     -- Power failure
     SIGMT,      -- Multitasking wake-up signal
     SIGMTKILL,  -- Multitasking kill signal
     SIGBUFIO,   -- Fortran asynchronous I/O completion
     SIGRECOVERY,-- Recovery signal (advisory)
     SIGUME,     -- Uncorrectable Memory Error
     SIGDLK,     -- True deadlock detected
     SIGCPULIM,  --
     SIGSHUTDN,  -- System shutdown imminent (advisory)
     SIGCRAY4,   -- Reserved for Cray Research, Inc.
     SIGRPE,     -- Register Parity Error
     SIGCRAY2,   -- Reserved for Cray Research, Inc.
     SIGCRAY1,   -- Reserved for Cray Research, Inc.
     SIGCRAY0,   -- Reserved for Cray Research, Inc.
     SIGINFO);   -- Quota warning or limit reached.
```

(continued)

```
For Signal use (SIGHUP      => 101,
                SIGINT      => 102,
                SIGQUIT     => 103,
                SIGILL      => 104,
                SIGTRAP     => 105,
                SIGABRT     => 106,
                SIGERR      => 107,
                --SIGFPE    => 108,
                --SIGKILL   => 109,
                SIGPRE      => 110,
                SIGORE      => 111,
                SIGSYS      => 112,
                SIGPIPE     => 113,
                SIGALRM     => 114,
                SIGTERM     => 115,
                SIGUSR1     => 116,
                SIGUSR2     => 117,
                SIGCLD      => 118,
                SIGPWR      => 119,
                SIGMT       => 120,
                SIGMTKILL   => 121,
                SIGBUFIO    => 122,
                SIGRECOVERY => 123,
                SIGUME      => 124,
                SIGDLK      => 125,
                SIGCPULIM   => 126,
                SIGSHUTDN   => 127,
                SIGCRAY4    => 128,
                SIGRPE      => 129,
                SIGCRAY2    => 130,
                SIGCRAY1    => 131,
                SIGCRAY0    => 132,
                SIGINFO     => 148);
```

(continued)

```
  Type Mapping_States is ( Mapping_off, Mapping_On );

    -- Default is Mapping_Of for all signs.  The default
     -- is to pass the signal on so that a user may write
     -- their own signal handler.

Procedure Set_Signal_Mapping ( direction: in Mapping_States );

    -- Sets signal mapping for all signals to 'direction'.


Procedure Set_Signal_Mapping ( sig      : in Signal;
                               direction: in Mapping_States );

    -- Sets signal mapping for 'signal' to 'direction'.


Function Signal_Number return Natural;

    -- Returns the signal number of the signal that was raised.  Tha value
     -- returned is the UNICOS value defined in /usr/include/sys/signal.h.

Function Signal_Number return Signal;

    -- Returns the number of the signal that was raised.  This value
    -- is compared against Signal declared above.

UNICOS_SIGNAL: Exception;
  -- Exception that signals are mapped to if mapping is on.

UNICOS_SIGNAL_Not_Raised: EXCEPTION;
  -- Exception that is raised if it was not possible
  -- to raise UNICOS_SIGNAL.
end UNICOS_Signal_Support;
```

The following example shows signal handling with package `UNICOS_Signal_Support`. The handler is turned on only to catch `SIGORE`. All other signals will be caught in the default manner by UNICOS.

```
--
-- This causes a UNICOS exception ORE which is caught in one
-- exception block, and not in another.
--

with UNICOS_Signal_Support;
with Unchecked_Conversion;
with Text_IO;

procedure Sigtest is

  package SIG renames UNICOS_Signal_Support;
  package IO renames Text_IO;

  --
  -- this causes an ORE signal by referencing nonexistent memory
  --

  procedure Raise_ORE IS
    type Acctype is access Integer;
    function Int2Acc is new Unchecked_Conversion(Integer, Acctype);
    A: Acctype := Int2Acc(-1);
    I: Integer;
  begin
    I:= A.ALL;
  end Raise_ORE;

begin
  IO.Put_Line ("Exception Handling Test");
  IO.New_Line;

  --
  -- turn on signal mapping for ORE, raise an ORE, and catch it
  --
  Exception_Block_1:
  begin
    SIG.Set_Signal_Mapping(SIG.SIGORE, SIG.MAPPING_ON);
     Raise.ORE;
     IO.Put_Line("Test 1: FAILED - ORE didn't get raised");
```

(continued)

```
  exception
    when SIG.UNICOS_Signal =>
      if SIG.Signal_Number = 11 then
        IO.Put_Line("Test 1: PASSED (caught the ORE)");
      else
      IO.Put_Line("Test 1: FAILED (caught the wrong signal)")
      raise
    end if;
    when others =>
      raise;
  end Exception_Block_1;

  --
  -- turn off signal mapping for ORE, raise an ORE, and fail to
  -- catch it
  --
  Exception_Block_2
  begin
    SIG.Set_Signal_Mapping (SIG.SIGORE, SIG.MAPPING_OFF);
     Raise_ORE;
     IO.Put_Line("Test 2: FAILED (ORE didn't get raised)");
   exception
     when SIG.UNICOS_Signal =>
        if SIG.Signal_Number = 11 then
           IO.Put_Line ("Test 1: FAILED (caught the ORE)
        else
            IO.Put_Line("Test 1: FAILED (caught the wrong signal)");
            raise;
        end if;
      when others
        raise;
  end Exception_Block_2;

end Sigtest;
```

The following is the output from the previous program. Notice that with signal mapping on, as in the first exception handling block, the program catches and properly identifies the ORE signal that occurred. In the second exception block, signal mapping is turned off, and when the ORE occurs again the default UNICOS action is taken and you get a core dump.

```
Lxception Handling Test

Test 1: PASSED (caught the ORE)
Operand range error (core dumped)
```

Cray Ada does not allow the following UNICOS signals to be trapped:

```
SIGKILL (Signal 09)
SIGCLD  (Signal 18)
```

When any of the preceding signals are trapped, (except for kill which can not be trapped) the Ada runtime provides a traceback showing the signal captured and the general location at which the error occurred or was translated into an Ada exception. The following information shows the traceback from an unhandled UNICOS exception when signal mapping has been enabled:

```
>>> UNHANDLED EXCEPTION <<<

Exception: UNICOS_SIGNAL
Reason:     UNICOS signal #11 (SIGORE) Operand Range Error

raised in sec/sigtest_1.sigtest_1 at line 39
```

Exception handling from foreign languages 5.8.5

Exceptions that originate in a foreign language routine or that are propagated up the call chain in a foreign language routine are not propagated into the Ada caller. Exception messages are printed out with a complete traceback that specifies the address from which the exception occurred or from which the routine was called.

The program then terminates. It is recommended that an EXPORT routine have an exception handler to avoid the possibility that any exceptions will escape.

The following is an exception that occurred in a Fortran subroutine called from an Ada main program. The exception used the INTERFACE pragma. Notice that the exception location in the Fortran routine is given as an address rather than as a line number.

```
>>> Unhandleable exception raised in FORTRAN routine <<<

Exception: NUMERIC_ERROR

  raised in FORTRAN routine AF_F_FUN at or near address       161c

 .called from sec/ada.f.ada_f between lines 11 and 15
```

In the following example, an exception occurred in an Ada procedure that was called from a Fortran subroutine using EXPORT pragma. The Fortran subroutine was in turn called from an Ada main program using INTERFACE pragma. Notice that in the exception traceback, the calling location in the Fortran routine is given as an address rather than as a line number.

```
>>> Unhandled exception <<<

Exception: NUMERIC_ERROR

  raised in sec/afa_exported.afa_a_ptoc at line 15

  called from export interface routine at address     17075a
  called from FORTRAN routine AFA_F_UN at or near address       164d
  called from sec/ada_f_ada.ada_f_ada between lines 25 and 29
```

Exception handling with optimization 5.8.6

Sometimes, because of optimization, the exact line number, at which an exception occurs cannot be determined. The following Ada source code illustrates this.

```
WITH Text_IO;

PROCEDURE L_E IS

  SUBTYPE T IS Integer RANGE -100 .. 100;

  PACKAGE IO RENAMES Test_IO;
  PACKAGE IIO IS NEW Test_IO.Integer_IO(T);

  I, J, K, L, M : T;

BEGIN
  IIO.Get(I);

  J := I + I;     — this is line 15
  K := J + J;
  L := K + K;
  M := L + L;

  IO.Put("16 times" );
  IIO.Put(I,Width => 0);
  IO.Put( "is" );
  IIO.Put(M,Width => 0);
  IO.New_Line;
END L_E;
```

The program accepts a number from the input and then doubles it 4 times. This yields an ultimate result of 16 times the number. If you enter 4, it says `16 times 4 is 64`. However, because the type of the result is constrained to `-100 .. 100`, larger entered numbers yield exceptions.

With no optimization (and thus no scheduling), it gets a constraint error exception on line 15 if you enter `64`, on line 16 if you enter `32`, on line 17 if you enter `16`, and on line 18 if you enter `8`.

With optimization (including the scheduler) on, if any of the additions yields a result greater than 100, the program reports a constraint error of the following type:

```
>>> Unhandled exception <<<

Exception: CONSTRAINT_ERROR
Reason:     value vs range constraint

raised in sec/1_e.1_e between lines 15 and 18
```

Because of instruction scheduling, lines 15 through 20 may be scheduled such that it is not possible to determine the exact line number on which the error occurred.

Subroutine linkage model
5.9

This subsection describes the basic model used to support subroutine linkage and data addressing for Ada on CRAY Y-MP, and CRAY X-MP, and CRAY-2 systems, including register conventions, stack frame layouts, and parameter passing mechanisms. Most users do not need this degree of detail, but some may, particularly if portions of their applications are written in assembly language.

Because Ada subprograms can, in general, be recursive and reentrant, subprogram calls must be modeled with a mechanism involving the allocation of subprogram data and linkage information on a stack in common memory. To retain compatibility in the presence of cross-language calls, the stack grows from low to high addresses.

Implementing this stack model requires two dedicated address registers: a frame pointer register (`FP = A7`) and a stack pointer register (`SP = B66` on CRAY X-MP systems and `$LM00 + 2` on CRAY-2 systems). The FP register points to the base of the active stack frame. Parameters, local data, and other data saved on subroutine entry are addressed by positive offsets from FP. The SP register marks the top of the stack. It is incremented and decremented during the entry and exit of a subprogram and when certain dynamically sized objects are allocated and deallocated. The FP register marks the base of the new stack frame on entry to a subroutine. It is updated to reflect the allocation of all local data required by the called subroutine, including parameters and various subroutine linkage pointers.

Before calling a subprogram, all parameters are loaded. On the CRAY Y-MP and CRAY X-MP systems, only the first four scalar and first four address parameters are loaded into B/T registers. The rest are passed on the stack in an overflow block. On

CRAY-2 systems, they are all loaded into local memory at `$LM00+8`. On entry to the called routine, the CRAY Y-MP and CRAY X-MP routines will store the parameters that were passed in auxiliary storage to the base of the new stack frame. On the CRAY-2 systems, if there is room in the frame package and none of the parameters are used up-level or referenced by address, then the parameters will be moved to the frame package. Otherwise, they will be stored in common memory in the new stack frame. FP and SP are updated to reflect the allocation of the new frame. As part of the return sequence, the subprogram restores the FP and SP registers to the values they had before the call.

Calling sequence for CRAY Y-MP and CRAY X-MP systems 5.9.1

On CRAY Y-MP and CRAY X-MP systems, the stack frame layout for a subprogram immediately after completion of its entry sequence is as shown in Figure 1.

	:	Low addresses
Frame pointer (FP) →	Link to TNB	(stkB77)
	Return address	(stkB00)
	Pointer to caller's parameter overflow block	(stkB01)
	Link to caller's frame base	(stkB02)
	Mark set pointer	(stkB03)
	Saved global display pointer	(stkB04)
	A1_Save	(stkB05)
	:	
	A6_Save	(stkB12)
	Address parameter 1	(stkB13)
	Address parameter 2	(stkB14)
	Address parameter 3	(stkB15)
	Address parameter 4	(stkB16)
	Address of parameter overflow block	(stkB17)
	B register save area	(stkB20-stkB24)
	S1_Save	(stkT00)
	:	
	S7_Save	(stkT06)
	Scalar parameter 1	(stkT07)
	Scalar parameter 2	(stkT10)
	Scalar parameter 3	(stkT11)
	Scalar parameter 4	(stkT12)
	T register save area	(stkB13-stkT17))
	Local variables	
Stack pointer (SP) →	:	High addresses

Figure 1. Subprogram stack frame layout on CRAY X-MP systems after completion of entry sequence

The following list shows the basic actions (in outline) that are performed for subprogram call, entry, exit, and return on CRAY X-MP systems:

Within the calling subprogram the following events occur:

1. Load the value of each parameter into a B or T register, as appropriate. Address parameters are loaded into B registers (starting with B13) and scalar parameters are loaded into T registers (starting with T07). Only the first four address and the first four scalar parameters are passed in auxiliary registers. After that, parameters will be stored on the stack. B17 will point to the word following the last parameter. This means that the last parameter would be accessed at B17–1.
2. If an elaboration check is required, check the elaboration Boolean. If it is false, raise ELABORATION_ERROR.
3. Perform a return jump to the entry point for the destination subprogram.

On entry to the called subprogram, the following actions occur:

1. Set FP to SP (new frame base is set to current top of stack).
2. Save the A and S registers that the subprogram will use in B05 .. B12 and T00 .. T06. Notice that A0, A7, and S0 need not be saved here.
3. Allocate the new frame by setting SP to FP + `frame_size`. `frame_size` is the sum of the storage requirements for local variables, as well as the largest parameter overflow block that this routine needs to build.
4. Initialize B77 with the pointer to the subprogram's Traceback Name Block (TNB).
5. If the subprogram's lexical level is less than the global display size and this subprogram has children that might reference its data up-level, then save `global_display(LL)` in B04. (This global display is in B35 .. B64.)
6. Save the B and T registers to the stack, starting at B77/T00 and ending at the last B/T register that will be used by the subprogram (for parameter lists and local data). This is generally B24 and T17. This has the effect of saving the TNB pointer, the return address, the static link, the caller's FP, the saved global display point, the saved A and S registers, and the parameters on the stack. It also saves any further B/T

registers the subprogram will use for parameter lists and local data.

7. Compare SP to the stack limit (contained in B67). If the limit is exceeded, call runtime support to extend the stack or raise the exception `STORAGE_ERROR` if the stack cannot be extended.
8. If this subprogram has children that might reference its data up-level, save FP in `global_display` (LL).
9. Save the new FP in B02.
10. Save B17 in B01. B01 will now contain the pointer to the parameter overflow block.
11. Copy SP into B17. This is where parameters for routines this routine calls will be put.

On exiting from the called subprogram the following actions occur:

1. If the subprogram is a function, move the function return value into S1.
2. Restore B00 through B24, T00 through T17. This has the effect of restoring the subprogram linkage, the first few parameters, and the auxiliary registers used for local data allocation.
3. If the subprogram's lexical level is less than the global display size and a display pointer was saved on entry to the subprogram, the restore `global_display` (LL) from B04.
4. Restore the saved A and S registers from the newly restored B and T registers.
5. Restore SP from FP.
6. Restore FP from B02.
7. Jump back to calling subprogram by using B00.

On returning to the calling subprogram, check and copy any scalar access parameters of mode `out` and of mode `in out` back into the appropriate variables. If the subprogram called was a function, then use the result from S1.

Calling sequence for CRAY-2 systems 5.9.2

On CRAY-2 systems, the stack frame layout for a subprogram immediately after completion of its entry sequence is as shown in Figure 2.

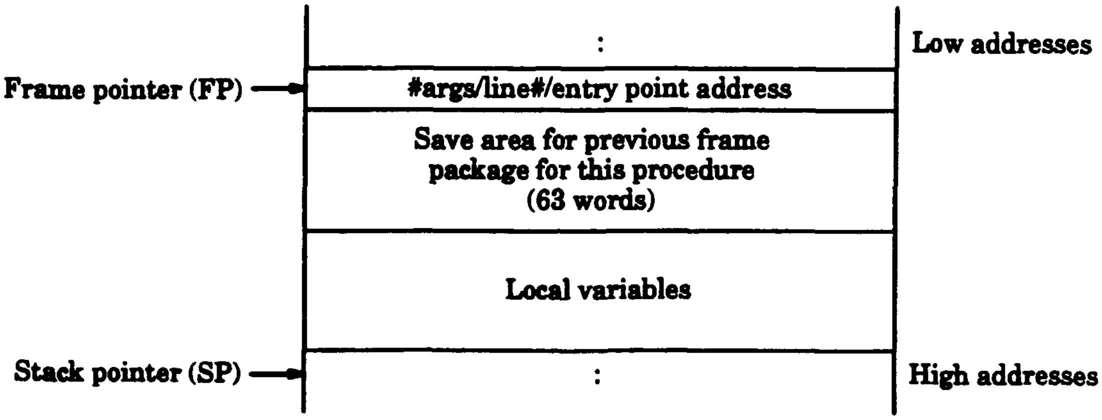

:
Low addresses
Frame pointer (FP)
#args/line#/entry point address
Save area for previous frame package for this procedure (63 words)
Local variables
Stack pointer (SP)
:
High addresses

Figure 2. Subprogram stack frame layout on CRAY-2 system after completion of entry sequence

The following lists show the basic actions performed for subprogram call, entry, exit, and return on CRAY-2 systems within the calling subprogram:

1. Load each parameter into local memory starting at `$LM00+8`.
2. Load S0 with the entry point of the destination subprogram, copy S0 to A0, and perform a return jump by using A0 (R,A0 A0). If an elaboration check is required, then check the elaboration Boolean and raise `ELABORATION_ERROR` if the called routine is not elaborated.

On entry to the called subprogram, the following actions occur:

1. Set FP to SP (new frame base is set to current top of stack).
2. Save A0, S0 and A1 ..A6, S1 .. S7 in `$LMADA`.
3. Allocate the new frame by setting SP to FP + `frame_size`.
4. Compare SP to the stack limit (contained in `$LM00+1`). If the limit is exceeded, call runtime support to either extend the stack or raise the exception `STORAGE_ERROR` if the stack cannot be extended.
5. Set `$LM00` to point to this frame package (`$FPK`).
6. If this is a recursive call to a subprogram that is already active in the current call chain, save S0 and the current contents of the subprogram frame package in the stack starting at FP–1.
7. If this is not a recursive call, save S0 (`#args/line#/entry point`) in the stack at FP–1.
8. Store return address, FP, and address of caller's frame package in the frame package. If this is a recursive call (that is, if you just saved the frame package in the stack frame), also store the frame base of the owner of the saved frame package in the frame package.
9. Save caller's A's and S's in the frame package. They are currently in `$LMADA`.
10. If necessary, store previous global display pointer for this lex-level in `$FPK+4`, and overwrite it with the value in A7.
11. Zero the mark set pointer at `$FPK+3`.

12. Move parameters from $LM00+8 to frame package or stack frame, depending on whether any of them need to be in common memory.

13. Save the address of the subprogram's frame package in $FPK+2.

On exiting from the called subprogram the following actions occur:

1. If this is a function then move the function return value into S1.

2. Restore the return address into A0 from the frame package.

3. Store the address of the caller's frame package in $LM00+0.

4. If necessary, restore previous global display from $FPK+4.

5. If there were any OUT or IN OUT parameters, copy them from either the stack or the frame package back to $LM00+8.

6. Restore caller's A and S registers from my frame package.

7. If this was a recursive routine (in other words, there is a frame package saved in the stack frame) then restore the frame package from the stack. If this was not a recursive routine, mark the frame package as inactive by zeroing the top half of the first word of the frame package (the pointer to the previous owner of the frame package).

8. Restore SP from FP.

9. Jump back to calling subprogram by using A0.

On return to the calling subprogram, the following actions occur:

1. Restore FP from $FPK+2. (This is done here, rather than in the exit sequence, because if there was a stack overflow in the called subprogram, the stack segment release performed as part of the return will destroy the contents of FP.)

2. Check and copy any scalar and access parameters of mode out and in out from local memory back in to the appropriate variables. If the subprogram called was a function, use the result from S1.

Parameter passing in Ada 5.9.3

This subsection discusses parameter passing in Ada, common to CRAY Y-MP, CRAY X-MP and CRAY-2 systems.

Parameters in Ada are passed either by value or by address, depending on the data type of the parameter. For parameters of array and record types, the address of the actual parameter is passed. Parameters of scalar and access types are passed by value (even for parameters of modes `out` and `in out`). In the case of scalar and access parameters of modes `out` and `in out`, the called routine must store the output values for such parameters back in the same locations used to pass them (either in B and T registers, the stack, or in the `$LM00` area) so that the calling routine can retrieve these values on return. The storage order of parameters corresponds to the order of formal parameter declarations in the Ada subprogram declaration. For CRAY X-MP systems, parameters passed by address or access types passed by value are passed in B registers, starting with B13. All other parameters are passed in T registers, starting with T07.

For calls to subroutines in other languages, the calling conventions of the target language are obeyed. (See "Interfacing to Other Languages," page 133, for more information on cross-language calls and restrictions on parameter passing.)

For CRAY Y-MP and CRAY X-MP systems, the first four parameters that are passed by address or are access types are passed in B registers, starting with B13. The rest are passed on the stack, indexed at negative offsets from B17. The first four scalar parameters are passed in T registers, starting with T07. The rest are passed on the stack, indexed at negative offsets from B17. For CRAY-2 systems, all parameters are passed in local memory, starting at `$LM00+8`.

Library Interfaces [6]

Page

Table

Library Interfaces [6]

Cray Ada supports a set of library interface routines which provide support directly from Cray Ada to basic math and utility routines. These routines are available in the following two packages:

- CRAY_LIB
- SYSTEM_INFO

Package CRAY_LIB 6.1

The routines provided in CRAY_LIB provide transparent efficient interfaces to logarithmic, trigonometric, bit manipulation and various utility routines. The compiler has special knowledge of these routines thereby allowing for vectorization of these routines to occur within Cray Ada code. CRAY_LIB contains three generic packages which are MATH_LIB, BIT_LIB, and UTIL_LIB with pre-instantiations as follows:

```
PACKAGE Cray_Math_Lib IS NEW Cray_Lib.Math_Lib(Float, Integer);
PACKAGE Cray_Bit_Lib IS NEW Cray_Lib.Bit_Lib(Integer);
PACKAGE Cray_Util_Lib IS NEW Cray_Lib.Util_Lib(Float, Integer);
```

The specification for the package CRAY_LIB is found beginning in the following discussion.

The routines in the CRAY_LIB package are actually interfaces to existing Cray library routines written in Cray Assembly language and Fortran. Additional details on these routines can be found in *Volume 3: UNICOS Math and Scientific Library Reference Manual*, publication SM–2081. By default, the only errors that are trapped in these routines are UNICOS floating_point (SIGFPE) signals. These are mapped to numeric_error. If you wish to trap any other signals while doing processing in the library routines, you must use the UNICOS_SIGNAL_Support package and specify the signals you want to handle.

The following is the specification for the CRAY_LIB package:

```
PACKAGE Cray_Lib IS

GENERIC

    TYPE Flt IS DIGITS <>;
    TYPE Int IS RANGE <>;

  PACKAGE Math_Lib IS

      FUNCTION  "**"(Val : IN Int;
                      Power : IN Int) RETURN Int;
      FUNCTION  "**"(Val : IN Flt;
                      Power : IN Int) RETURN Flt;
      FUNCTION  "**"(Val : IN Flt;
                      Power : IN Flt) RETURN Flt;

      FUNCTION Sqrt(Val : IN Flt) RETURN Flt;

      FUNCTION Log(Val : IN Flt) RETURN Flt;
      FUNCTION Log10(Val : IN Flt) RETURN Flt;
      FUNCTION Exp(Val : IN Flt) RETURN Flt;

      FUNCTION Cos(Val : IN Flt) RETURN Flt;
      FUNCTION Sin(Val : IN Flt) RETURN Flt;
      FUNCTION Tan(Val : IN Flt) RETURN Flt;
      FUNCTION Cot(Val : IN Flt) RETURN Flt;

      FUNCTION Acos(Val : IN Flt) RETURN Flt;
      FUNCTION Asin(Val : IN Flt) RETURN Flt;
      FUNCTION Atan(Val : IN Flt) RETURN Flt;
      FUNCTION Atan2(Val : IN Flt;
                     Val2 : IN Flt) RETURN Flt;

      FUNCTION Cosh(Val : IN Flt) RETURN Flt;
      FUNCTION Sinh(Val : IN Flt) RETURN Flt;
      FUNCTION Tanh(Val : IN Flt) RETURN Flt;

PRIVATE
  ...


END Math_Lib;
```

(continued)

```
GENERIC

    TYPE Int IS RANGE <>;

 PACKAGE Bit_Lib IS

      SUBTYPE Word_Range IS Natural RANGE 0 .. 63;

      SUBTYPE Word IS Integer;

      FUNCTION Leadz(Item : IN Word) RETURN Int;
      FUNCTION Popcnt(Item : IN Word) RETURN Int;
      FUNCTION Maskl(Val : IN Int) RETURN Word;
      FUNCTION Maskr(Val : IN Int) RETURN Word;
      FUNCTION Shiftl(Item : IN Word;
                     Val : IN Int) RETURN Word;
      FUNCTION Shiftr(Item : IN Word;
                      Val : IN Int) RETURN Word;

 PRIVATE

  ...

  end Bit_Lib;
```

(continued)

```
GENERIC

        TYPE Flt IS DIGITS <>;
        TYPE Int IS RANGE <>;

PACKAGE Util_Lib IS

        TYPE RTC_range IS RANGE 0 .. 35_184_372_088_831;

        -- 0 to 2**45-1.

        -- RTC_Float_Range is from 0 to 2**64-1.
        TYPE RTC_Float_Range IS DIGITS 10 RANGE 0.0..18_446_744_073_709_551_615.0;
        --
        -- type of a real-time clock starting time (for RTC_Timer_Start
        -- and RTC_Timer_Elapsed)
        --
        TYPE RTC_Value IS LIMITED PRIVATE;

        --
        -- type returned by Ranget and accepted by Ranset
        --
        TYPE Seed IS LIMITED PRIVATE;

        -- Floating point

        FUNCTION Ranf RETURN Flt;

        FUNCTION Sign( Val1: IN Flt;
                       Val2: IN Flt) RETURN Flt;


        -- integer

        --
        -- get a random seed
        --
        FUNCTION Ranget RETURN Seed;

        --
        -- set the random number generator's seed
        --
```

(continued)

```
PROCEDURE Ranset (Val: IN Seed);

--
-- convert an Int to a Seed
--
FUNCTION Int_To_Seed(I: IN Int) RETURN Seed;

--
-- the 'IMAGE and 'VALUE of a random number generator seed
--
FUNCTION Seed_Image(S: IN Seed) RETURN String;
FUNCTION Seed_Value(S_Image: IN String) RETURN Seed;

FUNCTION Sign(Val1 : IN Int;
              Val2 : IN INT) RETURN Int;
FUNCTION Trunc(Val1 : IN Flt) RETURN Int;

--
--real-time clock elapsed time routines which avoid problems with
--integer overflow; the RTC_Timer_Elapsed function enforces a limit
--of Int'LAST on its return value, and raises Constraint_Error if
--the result is too big (with type Integer, this is sufficient for
--(((2**45) -1) / (1000000000 / CT)) / 3600 hours, which is about
--1.6 days on a 4 ns CRAY-2 and 2.5 days on a 6 ns CRAY Y-MP)
--
PROCEDURE RTC_Timer_Start(RTC_Start_Time: OUT RTC_Value):
FUNCTION RTC_Timer_Elapsed(RTC_Start_Time: IN RTC_Value) RETURN Int;

-- Version 1 - The legal range of this version is 46 bits, however
--             this value may be exceeded within 2 to 3 days depending
--             on the system.  This routine actually returns a 64 bit
--             value.  In order to make use of this value checks must be
--             turned off.
FUNCTION RTC RETURN RTC_Float_Range;

-- Version 2 - Return RTC as a floating point value.
--             This is an actual Ada routine, not an external one.
FUNCTION RTC RETURN RTC_Range;
```

```
      -- Version 3 - Return clock ticks as a microsecond value in integer.
      --             This is an actual Ada routine, not an external one.

      FUNCTION Cpu_Usec RETURN Int;

PRIVATE

 ...

END Util_Lib;


END Cray_Lib;
```

CRAY_LIB contains several internal routines beginning with the alphanumeric Q8. It is best to avoid routines beginning with these values.

Table 7 summarizes the `CRAY_LIB` functions available with Cray Ada 2.0. The following lists define the conventions used in Table 7.

The level of vectorization is as follows:

F	Full vectorization
N	No vectorization

The type of code generated is as follows:

E	External
I	Inline
L	Language supported

In the Definition column, y is the function's result and the x values are function arguments. The following example shows this:

```
   A := SQRT(B)          K := ATAN2(X,Y)
```

Square brackets indicate the truncation of a term. If x has a value of 5.67, [x] equals 5.

Data types shown in the Function and Arguments columns are the following:

I	Integer
F	Float

Table 7. CRAY_LIB function summary

Purpose	Definition	Function		Arguments			Codes
		Name	Type	No	Type	Range	
Natural logarithm	y=ln(x)	`log`	F	1	F	0<x<infinity	F E
Common logarithm	y=log(x)	`log10`	F	1	F	0<x<infinity	F E
Square root	y=sqrt(x)	`sqrt`	F	1	I,F	0<=x<=infinity	F E
Exponent	y=exp(x)	`exp`	F	1	I,F	$\lvert x\rvert <2^{13}$ ln2	F E
Sine	y=sin(x)	`sin`	F	1	F	$\lvert x\rvert <2^{24}$	F E
Cosine	y=cos(x)	`cos`	F	1	F	$\lvert x\rvert <2^{24}$	F E
Tangent	y=tan(x)	`tan`	F	1	F	$\lvert x\rvert <2^{24}$	F E
Cotangent	y=cot(x)	`cot`	F	1	F	$\lvert x\rvert <2^{24}$	F E
Arccosine	y=arccos(x)	`acos`	F	1	F	$\lvert x\rvert$ <= 1	F E
Arcsine	y=arcsin(x)	`asin`	F	1	F	$\lvert x\rvert$ <= 1	F E
Arctangent	y=arctan(x) y=arctan(x1,x2)	`atan` `atan2`	F F	1 2	F F	$\lvert x\rvert$ < infinity $\lvert x1\rvert$ < infinity $\lvert x2\rvert$ /= 0	F E F E
Hyperbolic cosine	y=cosh(x)	`cosh`	F	1	F	$\lvert x\rvert <2^{13}$ ln2	F E
Hyperbolic sine	y=sinh(x)	`sinh`	F	1	F	$\lvert x\rvert$ <2 ln2	F E
Hyperbolic tangent	y=tanh(x)	`tanh`	F	1	F	$\lvert x\rvert$ < 2 ln2	F E
Truncation	y=[x] No rounding	`trunc`	I	1	F	$\lvert x\rvert < 2^{46}$	F I
Transfer sign	if x2 >=0 y:= x1,x2>=0 y:= –x1, x2< 0	`sign`	F	1	F	$\lvert x\rvert$ < infinity	F I
Leading zero count §	Counts number of leading 0 bits.	`leadz`	I	1	I		F I
Population count	Counts number of bits set to 1.	`popcnt`	I	1	I		F I

§ `leadz` vectorizes on CRAY-2 systems.

Table 7. CRAY_LIB function summary
(continued)

Purpose	Definition	Function		Arguments			Codes
		Name	Type	No	Type	Range	
Shift left	Shift x1 left x2 positions; leftmost positions lost; rightmost positions set to zero.	shiftl	I	2	I	0<= x2 <64	F I
Shift right	Shift x1 right x2 positions; rightmost positions lost; leftmost positions set to zero.	shiftr	I	2	I	0 <= x2 <64	F I
Mask left	Left-justified mask of x bits.	maskl	I	1	I		F I
Mask right	Right-justified mask of x bits.	maskr	I	1	I		F I
Exponentiation §	$y=x1^{x2}$ x1 is raised to the x2 power.	**	I,F	2	I,F		F E
Random number generator §§	y := r in which r is the first or next in a series of random numbers (0.0 ≤y ≤1.0).	ranf	F	0			F E
Get random seed	Gets the current random number seed.	ranget	I	0	I	|x| < infinity	N E
Set random seed §§§	Sets the current random number seed.	ranset	I	1	I	|x| < infinity	N E

§ Exponentiations where x1 is an integer and x2 is a float will not vectorize

§§ If ranf is called more than once in a vectorized loop it may produce random results in a different order than scalar mode.

§§§ See *Volume 3: UNICOS Math and Scientific Library Reference Manual*, publication SR–2081, for details.

Table 7. CRAY_LIB function summary (continued)

Purpose	Definition	Function		Arguments			Codes
		Name	Type	No	Type	Range	
Elapsed real time	Determine the elapsed real time between two events.	RTC_Timer_Start RTC_Timer_Elapsed	 I	1 1	I I		N L N L
Elapsed real time	Determine the elapsed real time between two events.	RTC_Timer_Start RTC_Timer_Elapsed	 I	1 1	I I		N L N L
Real-time clock	Returns the real-time clock value in a 46-bit integer or as a floating-point value.	rtc	I,F	0			N L
Clock ticks	Clock ticks in microseconds as an integer value.	cpu_usec	I	0			N L

The following is an example showing how to reference CRAY_LIB from an Ada program.

```
WITH Cray_Lib;
PROCEDURE Use_Cray_Lib IS
  --
  -- instantiate each package whose routines will be used
  --
PACKAGE Math_Lib IS NEW Cray_Lib.Math_Lib(Float, Integer);
PACKAGE Bit_Lib IS NEW Cray_Lib.Bit_Lib(Integer);
PACKAGE Util_Lib IS NEW Cray_Lib.Util_Lib(Float, Integer);

FUNCTION "***"              (Value: IN Float; Power: IN Integer)
                            RETURN Float RENAMES Math_Lib."***";
FUNCTION Popcnt             (Value: IN Bit_Lib.Word)
                            RETURN Integer RENAMES Bit_Lib.Popcnt;
FUNCTION Ranf               RETURN Float RENAMES Util_Lib.Ranf;
FUNCTION Int_To__Seed       (Value: IN Integer)
                            RETURN Util_Lib.Seed
                            RENAMES Util_Lib.Int_To_Seed;
PROCEDURE Ranset            (Value: IN Util_Lib.Seed)
                            RENAMES Util_Lib.Ranset;

  A: Float;
BEGIN
  Ranset (Int_To_Seed(12345));
  A := Ranf;
  FOR I IN 1 .. 4 LOOP
     A := (A ** I) + Float(Popcnt(I));
  END LOOP;
END Use_Cray_Lib;
```

System_Info package
6.2

The System_Info package may be used to identify which compiler was used to compile a program. The package interface is the following:

```
Package System_Info is

  Compiler_Version_Reference_Number: Constant := 2.0;
  Compiler_Version                 : Constant String := "2.0";

end System_Info:
```

An example showing use of the package interface is the following:

```
PACKAGE Compiler_Identification IS

  PROCEDURE Report_Compiler_ID;
END Compiler_Identification;

WITH System_Info;
With Text_IO;
PACKAGE BODY Compiler_Identification IS

  PROCEDURE Report_Compiler_ID IS

  BEGIN

    Text_IO.Put("The compiler ID is");
    Text_IO.Put(System_Info.Compiler_Version);
    Text_IO.New_Line;

  END Report_Compiler_ID;

END Compiler_Identification;
```

Interfacing to Other Languages [7]

Page

Page

Tables

Interfacing to Other Languages [7]

The Cray Ada Environment supports interfaces to other languages, as discussed in subsection 13.9 Interface to other languages of the *Reference Manual for the Ada Programming Language* (LRM). This means that routines written in Fortran, C, Pascal, and CAL can be called directly from Ada if they meet the following restrictions:

- Pragma `INTERFACE` can be applied only to subprograms for which users could have provided bodies. Examples of invalid pragma `INTERFACE` subprograms/names include names used as enumeration literals, attribute names, predefined operators, derived subprograms, and package bodies.
- In the case of overloaded subprogram names, pragma `INTERFACE` is allowed to stand for several subprograms. However, only subprograms declared earlier in the same declarative part or package specification are satisfied.
- If pragma `INTERFACE` is accepted and is applied to certain subprograms, it is invalid to provide a body for any of those subprograms, whether the pragma appears before or after the body.
- If the subprogram specified in pragma `INTERFACE` was declared by a renaming declaration, the pragma applies to the denoted subprogram, but only if the denoted subprogram otherwise satisfies the requirements.
- Nesting of pragma `INTERFACE` calls is supported. The level of nesting is limited only by available memory.

As noted in the LRM, all communication between foreign language routines and the Ada program must be achieved by the use of parameters and function results, and the subprograms must be as follows:

- Described by an Ada subprogram specification in the calling program
- Specified with an appropriate pragma `INTERFACE` directive in the calling program

In addition, the routines must be assembled or compiled by a language processor having data representation, calling, and runtime conventions that are compatible with the forms supported by the Cray Ada Environment, and whose object-file format is compatible with the UNICOS linker.

Users must ensure that any foreign object modules linked with Ada routines follow the same restrictions for addressing and code generation as those used in compiling the Ada modules. Cray Ada supports neither a mechanism for CPU targeting nor the UNICOS `target` command.

Currently, access to the following Cray Research system languages is supported from Cray Ada (see Table 8):

Table 8. Languages with interfaces supported in Ada

Language/interface	Language name
CF77, CFT	Fortran
C, UNICOS	C
CAL	Cray Assembly Language
Pascal	Pascal

No industry-wide standards exist for interlanguage programming. The conventions described in this manual apply only to software running on Cray Research systems. You can find descriptions of these conventions in *Interlanguage Programming Conventions*, publication SN–3009.

Ada also supports callback. A foreign routine can call an Ada routine by using the pragma EXPORT. In this case, the main program must be Ada. An additional restriction is that the foreign language caller must not include multitasking. See "Calling Ada from foreign languages," page 160.

See "Exception handling from foreign languages," page 106.

Data types
7.1

All the data structures defined within any one of the languages listed in the *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014, cannot necessarily be defined in the other languages. Some restrictions result from intrinsic characteristics of the languages and some result from the particular implementations. The rules that follow are for data mappings on Cray Research computer systems and do not necessarily apply to other vendors' implementations.

The only data types having values that can be shared directly by all languages previously listed are as follows:

- 64-bit integer
- Single-precision floating point
- One-dimensional arrays of 64-bit integers
- One-dimensional arrays of single-precision floating point

Data types that can be shared between these languages, but only with some special processing, are as follows:

- Character strings
- Boolean (logical) values
- Multidimensional arrays of 64-bit integer, single-precision floating point, and Boolean values
- Word pointers to values of 64-bit integer, single-precision floating point, or logical values, or to arrays of those types of values

Cautions
7.2

Remember the following cautions when using pragma `INTERFACE`:

- Setting your own error conditions, such as floating-point exceptions, while in a foreign language module can invalidate Ada. If this is done, the integrity of the remaining Ada execution cannot be ensured as correct and complete.
- Do not call `sbreak`; use `malloc` instead. The use of `sbreak` may cause unpredictable results.

- Currently, only one language at a time should perform I/O. If you write an Ada program that calls a C routine and both Ada and C modules attempted I/O, the result from Ada may be erroneous I/O or no I/O at all. This is because of the difference in the way output file identifiers are mapped between various languages.
- Pragma INTERFACE is portable only on Cray Research systems.
- Only scalars and access types can be returned to Ada modules from foreign module function calls.
- Ada, Fortran, C, and Pascal all provide ways of representing strings of characters, but the semantics of these character-string types vary a great deal among languages and Cray Research systems. Details about this are beyond the scope of this manual. See *Interlanguage Programming Conventions*, publication SN–3009.
- Foreign code segments may be debugged only at the machine level if you use adbg. At the machine level, you can also use CDBX.

Calling and parameter-passing conventions 7.3

Two different calling and parameter-passing conventions are supported by the Cray Ada Environment: an internal format and UNICOS standard format.

The C and UNICOS format is specified with the following statement:

```
pragma INTERFACE (C, subprogram_name)
```

An interface using the name "UNICOS" is not supported. The pragma INTERFACE to C is intended for interfacing to code generated by C language compilers, or by any other language compilers that follow UNICOS conventions. It is discussed in detail in *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

When interfacing to either Fortran or Pascal, the same pragma interface specification is defined as for both Cray Assembly Language and C, except that the appropriate language identifier is substituted. Each of these language interfaces is discussed in detail later in this section.

After the Ada units are compiled and the foreign language code is compiled and assembled, the combined code must be bound and linked. This is accomplished by using the `ada` command with the `-m` option (see *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014) or by using the `ald` command (see *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014).

General interfacing considerations
7.4

The Ada name of the designated subprogram is referenced directly as a global symbol; it must resolve to an identical symbol defined at link time, presumably by the foreign language routine to be called. One implication of this fact is that the name of the foreign routine must conform to Ada identifier rules (such as starting with a letter, containing only letters, digits, or underscore characters, and so forth). Another is that pragma `INTERFACE` cannot be used for overloaded subprograms.

These name restrictions may be circumvented by using one of the Cray Ada Environment implementation-defined pragmas: `LINKNAME` or `INTERFACE_INFORMATION`.

Either pragma must be specified immediately following an `INTERFACE` pragma. These pragmas cannot apply to multiple overloaded subprograms (unlike pragma `INTERFACE`).

Note

Pragmas `LINKNAME` and `INTERFACE_INFORMATION` are supported in Cray Ada 2.0. Pragma `INTERFACE_INFORMATION` provides all of the functionality of `LINKNAME` and adds some functions. `LINKNAME` is being retained for purposes of upward compatibility. When Cray Ada 3.0 is released, however, Cray Research will no longer support pragma `LINKNAME`.

Using both pragma `INTERFACE_INFORMATION` and `LINKNAME` against the same pragma `INTERFACE` call is illegal, because both are required to follow the pragma `INTERFACE` call immediately.

Pragma INTERFACE_INFORMATION 7.4.1

Pragma INTERFACE_INFORMATION takes three arguments. The first argument is a subprogram name that has been previously specified in pragma INTERFACE. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The third specifies the mechanism by which the routine will be called.

If pragma INTERFACE_INFORMATION is used, it must immediately follow pragma INTERFACE for the associated subprogram; otherwise a warning is issued, indicating that pragma INTERFACE_INFORMATION has no effect.

Linking many foreign object modules by using the ald command may become cumbersome. The user of a SEGLDR directives file may simplify the procedure. See *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014, for more information about linking foreign object modules.

The syntax of pragma INTERFACE_INFORMATION is as follows:

```
pragma INTERFACE_INFORMATION ( Name => subprogram_name,
                    [ [Link_Name =>] string_literal, ]
                    [ [Mechanism =>] "PROTECTED" | "UNPROTECTED" , ];
```

subprogram_name
: You must specify a *subprogram_name*. All other arguments are optional.

Link_Name=> *string_literal*
: Provides the link name to be used in generating calls to the subprogram. In the absence of this argument, the lowercase form of the subprogram's Ada name is used for the linkage. This argument supplants the functionality provided by the LINKNAME pragma.

[Mechanism =>] "PROTECTED" | "UNPROTECTED"
: Specifies whether calls to the given subprogram are protected against exceptions caused by the foreign interface routine. The default for this argument is "PROTECTED". In certain cases, the code generator must generate extra code to protect the runtime support from being confused because of the lack of standard trace name blocks (TNBs) in foreign code (particularly for calls to assembly).

An example of the use of pragma INTERFACE_INFORMATION is as follows:

```
procedure Do_Something(Addr: System.address; Len : Integer);
Pragma INTERFACE (C, Do_Something);
Pragma INTERFACE_INFORMATION (Name => Do_Something,
                              Link_Name => "CHANGE",
                             Mechanism => "UNPROTECTED");
```

Pragma LINKNAME
7.4.2

The syntax of pragma LINKNAME is as follows:

```
pragma LINKNAME (subprog_name, string_lit);
```

The following is an example of pragma LINKNAME:

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma INTERFACE (assembly, Dummy_Access );
pragma LINKNAME (Dummy_Access, "_access");
```

System users are urged to use pragma INTERFACE_INFORMATION instead of pragma LINKNAME.

Pragma EXPORT provides a means by which foreign code can make calls to Ada programs, provided that the main subprogram is an Ada routine. See "Calling Ada from foreign languages," page 160.

Interlanguage data access
7.4.3

A new function, label, in a package System has been added to Cray Ada. label provides for more convenient interlanguage access to data, namely, common blocks. This mechanism provides not for the actual declaration of common blocks but for the referencing of them.

It is not possible for a program composed of only Ada elements to declare and use common blocks. To use common blocks, a program must be composed of elements written in languages other than Ada as well as elements written in Ada.

The declaration of label is the following:

```
function label (Name:string) return Address;
```

The value of the Name parameter is the following:

```
"<linkname>,<linkage>,"
```

<linkage> is defined as one of the following:

External <linkname> is the name of a normal external variable.

Common The <linkname> is the name of a named Fortran block. If <linkname> is empty, blank common is referred to.

C-external The <linkname> is the name of an external C variable. This is actually a synonym for the <linkage> common because C external actually references the named Fortran blocks. The <linkage> name is the name of the common block.

For further information, see "Package system," page 203.

Pragma INTERFACE for assembly 7.5

One usage of pragma INTERFACE that is currently supported by the Cray Ada Environment is to call assembly language modules. Its syntax is as follows:

```
pragma INTERFACE (assembly, Ada_subprogram_name);
```

The calling conventions for the call to the assembly language routine are the same as those for a call to a Fortran-implemented routine. The restrictions for an interface to Fortran also apply.

Pragma INTERFACE for C and UNICOS
7.6

The Cray Ada Environment supports pragma INTERFACE to routines written in C and other languages that adhere strictly to C interface conventions. Because UNICOS system calls are written in C, this pragma works as an interface to UNICOS. Its syntax is as follows:

```
pragma INTERFACE (C, Ada_subprogram_name);
```

The rules for naming subprograms are the same as those for pragma INTERFACE to assembly; that is, the name of the C routine being specified in the INTERFACE pragma must be a legal Ada identifier (unless pragma INTERFACE_INFORMATION is used), and the name may not be overloaded. For example, given a pragma INTERFACE statement such as the following:

```
pragma INTERFACE (C, Get_Argument);
```

If the earlier version of C were used, one of the following two pragma Interface_Information statements would also be required:

```
pragma Interface_Information (Name => Get_Argument, "get$argument); -- X-MP
pragma Interface_Information (Name => Get_Argument, "get@argument); -- CRAY-2
```

C usage considerations
7.6.1

The following subsections include C underscore changes, C parameter passing, C string parameters, and C special global names.

C underscore changes
7.6.1.1

Earlier versions of the Cray Research C compiler (those prior to C 4.1) translated an '_' character to either a '$' character (on CRAY X-MP systems) or an '@' character (on CRAY-2 systems). If you call a C routine that is compiled using an older version of the Cray C compiler, you must have a pragma INTERFACE_INFORMATION statement.

C parameter passing 7.6.1.2

Because the C programming language specifies the passing of arguments strictly by value, only in arguments may reliably be passed to C functions. Although a C routine may legally include an assignment to a formal parameter, it is not guaranteed that the assignment will result in an update to the stack copy of the parameter on exit from the routine, as required for Adas' model for `out` and `in out` parameters. To pass values back to the calling program, you may specify that the value is returned through the function return mechanism. This permits the return of nonaggregate data types. If you want to return objects of other types, you may pass pointers to the objects in the calling program (such as arrays) in which the results are to be stored. The called routine can then access these objects through the C pointer mechanism.

The use of pragma `INTERFACE` to C limits the types of data that may be passed to a C subroutine. The types currently supported for parameters in C interface routines are scalar types (integer, enumeration, and floating point), access types, and type `system.address`. All parameters in the Ada subprogram declaration must have mode `in`. These data type restrictions on parameters also apply to the return types of functions that interface to C.

C string parameters 7.6.1.3

Strings in C, by convention, are null-terminated, and they are passed by the address of the first element. There are no implicit index values or lengths associated with a string, and it is up to the code that handles the string to test for the null character that terminates the string. Strings in Ada, on the other hand, carry implicit index values for the first and last elements, and they are not null-terminated.

The recommended (and most efficient) way to pass strings between Ada and C routines is for the Ada code to follow the C conventions explicitly. To pass a string to a C routine, the Ada code would store a null terminator at the end of the string and pass the address of its first element. A C function returning a string would be declared as an Ada function returning a value of type `system.address`. Only string values that start on word boundaries should be passed to C. It is safe to pass string variables and string literals as parameters, but not arbitrary string slices.

A specific example of string passing is provided in "Command-line argument example," page 146.

C special global names 7.6.1.4

The following global names have been declared in the run-time code specifically for use in C programs:

```
extern int rtargc;   /* Number of command line arguments*/
extern char **rtargv; /* Pointer to array of argument strings*/
extern char **environ;      /* Pointer to the environment variable string*/
```

The three variables are used in the same way as are the main program arguments argc, argv, and envp.

C usage examples 7.6.2

This subsection provides examples that explain the use of pragma INTERFACE to C. Table 9 shows a generalized mapping of Ada to C data types and the method used for passing them. This information is very system-dependent, is neither portable nor standard and is recommended only for specialized applications by knowledgeable users. There is no direct mapping for data types other than scalars, arrays of scalars, and access types. It is recommended that you contact your system support staff for information based on your specific Cray Research system and language environment.

Table 9. Summary of parameter types for Ada calling C

Parameters sent by Ada		Parameters received by C		Ada function
Type	**Passed by**	**Type**	**Passed by**	**Results**
integer	integer	int/short/long	value	integer
float	float	float	value	float
array	array(array'FIRST)'address	array	reference	Illegal§
access	access	pointer	value	access

§ You can use an access type to point to this structure.

Calling C library routines 7.6.2.1

This subsection is the first of two that show how pragma INTERFACE to C may be used in Ada applications. In this subsection, an Ada procedure, Random_Number, is provided as an example. Random_Number generates and prints a ranaom integer based on a user-entered seed value. It generates this number by making a direct call to C library functions Srand and Rand. No user-written C code is required, and users are not required to load library libc, because the Ada linker (ald command) does this by default.

```
--          Ada procedure Random_Number       --


with Text_Io;
procedure Random_Number is

  -- I/O
  package Iio is new Text_Io.Integer_Io (Integer);
  Number : Integer;

  -- Set up for pragma interface to Srand and Rand
  function Srand (Seed : Integer) return Integer;
  pragma Interface (C, Srand);
  function Rand return Integer;
  pragma Interface (C, Rand);

begin
  Text_Io.Put ("Enter seed for random number generator: ");
  Iio.Get (Number);
  Text_Io.New_Line;
  Number := Srand (Number);

  Number := rand;
  Text_Io.Put ("Random number is as follows: ");
  Iio.Put (Number);
  Text_Io.New_Line;
end Random_Number;
```

In the declarative part of the program, the specifications for Rand and Srand are defined in the usual fashion. The pragmas immediately follow the specifications. For information on the Ada rules for preparing pragma INTERFACE specifications, see subsection 13.9 of the LRM.

To make the Ada procedure Random_Number executable, compile and link the file random.ada, which contains the main program random_number. A copy of random.ada is in the /usr/lib/ada/examples directory. During linking, the C library is searched automatically for the Srand and Rand routines.

```
    ada -m random_number random.ada
```

Command-line argument example 7.6.2.2

This subsection provides another example of the use of pragma INTERFACE to C. This time, an Ada procedure, Show_Argument, calls C procedure Get_Argument to return command-line arguments. The C procedure uses global variables rtargc and rtargv.

The text of the calling Ada procedure follows; a copy of the procedure is also available in file show_args.ada in the /usr/lib/ada/examples directory. Note the use of C pointer types and the Ada system.address type to give the C routine access to the string object that is to contain the returned argument.

```
-- Ada procedure Show_Argument --

with System;
with Text_Io;
procedure Show_Argument is
  -- I/O
  package Iio is new Text_Io.Integer_Io (Integer);
  Position : Integer;
  Argument : String (1 .. 1000);
  Arg_Len  : Integer;

  -- Set up for pragma INTERFACE to Get_Argument
  function Get_Argument (Parameter_1 : Integer;
                         Parameter_2 : System.Address)
                         return Integer;

  pragma Interface (C, Get_Argument);

begin
  Text_Io.Put ("Enter position number of argument: ");
  Iio.Get (Position);
  Arg_Len := Get_Argument (Position, Argument(Argument'FIRST)'ADDRESS);
  Text_Io.Put_Line ("Argument is as follows: " & Argument (1 .. Arg_Len));

end Show_Argument;
```

The following is the text of the C routine called from Ada procedure Show_Argument:

```
/*---------------------------------------------------------------------*/
/*                -- C routine get_argument --                         */
/*---------------------------------------------------------------------*/

short get_argument (position, arg_ptr)

short  position; /* position number of argument to be returned */
char  *arg_ptr;  /* pointer to string in which to store argument */

{
  extern int rtargc;    /* number of command-line arguments */
  extern char **rtargv; /* pointers to command-line arguments */

  short strndx;  /* loop counter/string index */
  char  c;       /* temporary character */

  /* check argument position number*/
  if (position > rtargc - 1) return (0);

  /*one pass for every character in the parameter */
  /* until the null character at the end of the    */
  /* parameter is found                            */
  for (strndx = 0 ; c = rtargv[position][strndx] ; strndx++)
            arg_ptr[strndx] = c;

  return (strndx); /* return the length of the string */
}
```

Ada integer types have 46-bit precision and are passed as 64-bit quantities. Chosing a C data type of int, short, or long depends on the application. A short integer is used in the preceding example. See the *Cray Standard C Programmer's Reference Manual*, publication SR-2074, or the *Cray C Reference Manual*, publication SR-2024, for information about C integer types.

The steps involved in making the Ada procedure executable are as follows:

1. Compile the C routine. To do this, compile Get_Argument with the native C compiler, as in the following example:

   ```
   scc -c get_arg.c
   ```

   In this example, scc invokes the Cray Standard C compiler, and the -c option tells the compiler to compile the source code in file get_arg.c without linking it. The resulting object code is stored in the get_arg.o file. Remember to target your C code for an EMA system when compiling on a CRAY Y-MP or CRAY X-MP system.

2. Compile and link the calling Ada procedure. First invoke the Ada compiler and then the Ada linker, as in the following example:

   ```
   ada /usr/lib/ada/examples/show_args.ada
   ald -p 'get_arg.o' show_argument
   ```

   In this example, show_args.ada is the source file containing the Ada procedure Show_Argument, show_argument is the executable file the linker produces and puts in the current working directory, and get_arg.o is the object module produced by the C compiler. The -p option directs the compiler to include object file get_arg.o in the link. The -p option can also appear on the command line for ada when the -m option is used. In this case, the compiler passes the option to the linker. For details on the ada and ald commands, see *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

Accessing C global variables 7.6.2.3

The following example uses System.Label to access external variable errno. When errno is used, external must be specified in the declaration for System.Label.

```
with SYSTEM;
package LIBC_ROUTINE is

  -- Declaration for a LIBC routine used to create a file
--
function CREATE_FILE(FILENAME  : in SYSTEM.ADDRESS;
                      MODE: in INTEGER) return INTEGER;

pragma INTERFACE (C, CREATE_FILE);
pragma INTERFACE_INFORMATION (CREATE.FILE,"create");

end LIBC_ROUTINE;
-------------------------------------------------------------------
with System, UNCHECKED_CONVERSION, TEXT_IO, INTEGER_TEXT_IO;
with LIBC_ROUTINE;

procedure lab003 is
  -- an example of how to map global C variables to Ada variables.
  -- The LIBC routine "creat" creates a file in a directory.  If the
  -- file creation fails, the global C variable "errno" is updated
  -- with the reason for the failure.  In order for an Ada program to
  -- examine this global value, an address clause is used to map
  -- the global C variable to a local Ada object.

  ERROR1 : INTEGER;
  ERRNO : INTEGER;
  for ERRNO use at System.Label ("errno,external");

  FILE_CREATION_FAILED : exception;

  FILE_NAME : STRING (1 .. 14);
  SUCCESSFUL_CREATE : constant INTEGER := 0;  --
  FILE_DESCRIPTOR : INTEGER;
  LAST : INTEGER;

begin

  TEXT_IO.PUT ("Enter file name to create:");
  TEXT_IO.GET_LINE (FILE_NAME, LAST);
  FILE_DESCRIPTOR := LIBC_ROUTINE.CREATE_FILE (FILE_NAME'ADDRESS,
8#777#);
ERROR! := ERROR1;
```

(continued)

```
if FILE_DESCRIPTOR < 0 then
  raise FILE_CREATION_FAILED;
end if;

exception
  when FILE_CREATION_FAILED =>
    TEXT_IO.PUT ("File creation failed with error number: ");
    INTEGER_TEXT_IO.PUT (ERROR1);
    TEXT_IO.NEW_LINE;

end lab003
```

Pragma INTERFACE to Fortran

7.7

The Cray Ada Environment supports pragma INTERFACE to routines written in Fortran. The format for pragma INTERFACE to a Fortran routine is as follows:

```
pragma INTERFACE (Fortran, Ada_subprogram_name);
```

Fortran specifies the passing of arguments by address (reference). Therefore, the compiler will pass the address of parameters, unless the actual is an address or access types.

Functions may also be declared as being interfaced to Fortran, but their result type is restricted to being either a scalar type, access type, or system.address type (Ada access types correspond to Fortran pointer types).

Table 10 shows a generalized mapping of Ada to Fortran data types and the method used for passing them. This information is very system dependent, is neither portable nor standard, and is recommended only for specialized applications by knowledgeable users. There is no direct mapping for data types other than scalars, arrays of scalars, and access types. It is recommended that you contact your system support staff for information based on your specific Cray Research system and language environment.

Table 10. Summary of parameter types for Ada calling Fortran

Parameters sent by Ada		Parameters received by Fortran		Ada function
Type	**Passed by**	**Type**	**Passed by**	**Results**
`integer`	`integer'address`	`integer`	Reference	`integer`
`float`	`float'address`	`real`	Reference	`float`
`array(1..n,1..m)`	`array(array'FIRST)'ADDRESS`	`array`	Reference	Illegal §
`access`	`access'address`	`pointer`	Reference	`access`

§ You could use an access type to point to this structure.

Fortran usage considerations 7.7.1

The following subsections describe considerations specific to Fortran. These include both parameter passing and string parameters.

Fortran parameter passing 7.7.1.1

The interlanguage programming conventions described in this manual apply to Fortran. Ada programmers must transform data to conform to these conventions if they intend to write programs that may have to interface with either programs or library routines that may be written in other languages. (Many Cray Research library routines are written in Fortran.)

Fortran stores multidimensional arrays differently than do Ada, C, and Pascal; it keeps them in column-major order, because that is how Cray Research library routines operate on them. (UNIX library routines may be different than those in UNICOS.) Because of this difference between Fortran and Ada (as one example), routines that are written in Ada, declare multidimensional arrays, and that may be used with Fortran code or passed to library routines should be careful to do the following:

- Reverse the order of the dimensions in the declarations
- Reverse referencing subscripts in the code to conform to the column-major storage system

See the listing for `get_copy.ada` on page 156 for an example of this.

Fortran string parameters 7.7.1.2

Fortran passes strings by using a Fortran character descriptor. There is no direct mapping from an Ada character string to a Fortran character descriptor in Cray Ada release 2.0. Contact your local Cray Research representative for information specific to your system and language environment.

Fortran usage examples 7.7.2

This subsection provides several examples explaining the use of pragma `INTERFACE` for Fortran. The test cases do not necessarily do any useful work; they are meant to be simple, concise examples of how to pass parameters between Ada and Fortran. Source files for each example are in the `/usr/lib/ada/examples` directory.

Calling a Fortran library routine 7.7.2.1

In this example, an Ada program, `ada_sin.ada`, uses pragma `INTERFACE` to call a Fortran math library routine, `sin`. The Ada routine follows:

```
with System;
with Text_Io;

procedure Ada_Sin is

  -- I/O
  package Fio is new Text_Io.Float_Io
(Float);
  -- Define Variables
  X : Float;-- input value to sin
  Y : Float;-- return value from sin

  -- Set up for pragma interface to Sin
  function Sin (X : System.Address) return
Float;
  pragma Interface (Fortran, Sin);

begin
  X := 1.570795;
  Y := Sin (X'Address);
  Text_Io.Put ("sin of ");
  Fio.Put (X);
  Text_Io.Put (" is ");
  Fio.Put (Y);
  Text_Io.New_Line;
end Ada_Sin;
```

The Ada program can be compiled and linked by the following commands:

```
ada -v ada_sin.ada

ald -v ada_sin
```

The first of these two command lines calls the Ada compiler to compile ada_sin.ada. The ald command calls the Ada linker to bind and link the object files in the Ada library. The Fortran SIN routine is located in UNICOS library libm.a. Because this is one of the libraries that SEGLDR searches by default, it is not necessary to specify this library explicitly on the linkage command line. The executable module is called ada_sin.

Calling a user-defined Fortran routine 7.7.2.2

In the following example, an Ada program, `chng_flt.ada`, calls a Fortran routine, `change.f`, passing an integer value to it. The integer is changed to a floating-point number, multiplied by a floating-point value, and the result is passed back to the Ada routine, which then prints it. The Ada routine, `chng_flt.ada`, follows.

```
--    Send_value = integer;
--    Return_value = float;
with System;
with Text_Io;

procedure Chng_Flt is

  -- I/O
  subtype Flt is Float;
  subtype Int is Integer;
  package Fio is new Text_Io.Float_Io (Flt);
  package Iio is new Text_Io.Integer_Io (Int);

  -- Define Variables
  Send_Value : Int;
  Return_Value : Flt;

  -- Set up for pragma interface to Change
  function Change (Intarg : System.Address) return Float;
  pragma Interface (Fortran, Change);

begin
  Send_Value := 1025032;
  Return_Value := Change (Send_Value'Address);
  -- preceding line is call to Fortran
  Text_Io.Put ("Integer Value = ");
  Iio.Put (Send_Value);
  Text_Io.Put ("   Float Value = ");
  Fio.Put (Return_Value);
  Text_Io.New_Line;
end Chng_Flt;
```

The Fortran routine change is as follows:

```
C   returns 15.325 times the Input Value.
      function change(ival)
      integer ival
      change = float(ival) * 15.325
      return
      end
```

The example can be compiled and linked by the following commands:

```
cf77 change.f
ada -v -m chng_flt -p 'change.o' chng_flt.ada
```

The CF77 compilation uses its default parameters and produces an object file, change.o. Remember to target your Fortran code for an EMA system when compiling on a CRAY X-MP EA system. The Ada command line asks for verbose messages and automatically calls the Ada linker, producing an executable file, chng_flt. The -p option is sent to the linker, and it requests that the object module be included at link time. The chng_flt.ada file is the Ada module to be compiled.

Accessing Fortran common blocks 7.7.2.3

This subsection includes two examples. They are examples of accessing Fortran common blocks and external data. In the second example, the Ada routine copies the Fortran array.

The following is an example of using system.label. A common block is set up in Fortran. The Ada program calls the Fortran subroutine, and the Fortran subroutine assigns values to the variables in the common block. On return to Ada, the values are printed to show that the two routines are sharing the data.

To compile and link this code, perform the following:

```
ada -v lab001.ada
cf77 lab002.f
ald -v -p 'lab002.o' lab001
```

```
  with SYSTEM,TEXT_IO;

  PROCEDURE lab001 IS


  TYPE common_rec IS
    RECORD
     X:INTEGER;
     Y:INTEGER;
     Z:INTEGER;
  END RECORD;

  PRAGMA PRESERVE_LAYOUT(common_rec);

  comm_rec_var:common_rec;
   for comm_rec_var use at system.label("C1,COMMON");

   package Iio is new Text_Io.Integer_Io(Integer);
   procedure Lab002;
   pragma INTERFACE(Fortran, Lab002);

  BEGIN
    text_IO.Put_Line("test lab001");
    Lab002;
    Iio.Put(comm_rec_var.X);
    Iic.Put(comm_rec_var.Y);
  END;

Fortran code:

          SUBROUTINE LAB002()
          COMMON/C1/ix,iy,iz
          ix=5
          iy=6
          iz=7
           END
```

The previous example shows how to copy an array from a Fortran common block by using the Ada `get_copy.ada` routine, which calls a Fortran subroutine, `copy.f`. Because the array is copied, rather than shared as in the previous example, any assignments to the copied array in Ada do not affect the Fortran version of the array. The dimensions and order of the subscripts must be reversed, because Fortran stores array elements by column-major order.

The source codes for both of the routines in this example are located in `/usr/lib/ada/examples`.

The Ada routine, `get_copy.ada`, follows:

```
  with System;
  with Text_Io;
  procedure get_copy is

   -- Data
   X_copy: array(1..2,1..3) of Float;
   -- I/O
   package Fio is new Text_Io.Float_Io (Float);

   -- Set up for pragma interface to Copy
   procedure Copy(I: In System.Address);
   pragma Interface (Fortran, Copy);

begin
   Copy(X_copy'address);
   Text_Io.Put ("The values of array x are:  ");
   Text_Io.New_Line;
   for I in 1..2 loop
     for J in 1..3 loop
       Fio.Put (X_copy(I,J));
     end loop;
     Text_Io.New_Line;
   end loop;
   Text_Io.New_Line;

end get_copy;
```

The Fortran routine, copy.f, follows.

```
      subroutine copy(y)
C reverse dimensions
      common /block2/x(2,3)
      dimension y(3,2)
      data x/1.1,2.1,1.2,2.2,1.3,3.3/
      write (*,*) x(1,1),x(1,2),x(1,3)
      write (*,*) x(2,1),x(2,2),x(2,3)
C reverse subscripts
      do 10 i=1,2
      do 10 j=1,3
10    y(j,i) = x(i,j)
      return
      end
```

You can compile and link the preceding example, producing an executable file named get_copy, with the following UNICOS command lines:

```
cf77 /usr/lib/ada/examples/copy.f
ada -m get_copy -p 'copy.o' /usr/lib/ada/examples/get_copy.ada
```

Pragma INTERFACE for Pascal

7.8

The Cray Ada Environment supports pragma INTERFACE to routines written in Pascal. The format for a pragma that provides an interface to a Pascal routine is as follows:

```
pragma INTERFACE (Pascal, Ada_subprogram_name);
```

The Pascal routine must be declared Exported, as follows:

```
procedure Ada_subprog_name; Exported
(ada_subprog_name);
```

The Exported statement is necessary, and it must have as its argument the all-lowercase name of the routine as declared in the INTERFACE pragma. If pragma INTERFACE_INFORMATION is used instead of INTERFACE, the routine-name argument is specified exactly as declared in INTERFACE_.INFORMATION.

Pascal routines called from Ada are limited to having only value and VAR parameters that are scalars, pointers, composites (though no conformant arrays are permitted), or strings. Pascal functions may return only scalars or pointers.

To pass a variable to a Pascal VAR parameter, the Ada declaration of that Pascal routine must declare that parameter as in out. Pascal value parameters must be declared in Ada as in parameters. All other combinations are currently unsupported and can cause unpredictable results.

Ada composites passed to Pascal must be constrained. Ada strings passed to Pascal must be constrained and word aligned.

Note

If any of the parameters are access types, Pascal pointer checking must be disabled. Insert (*#RPN*) into your Pascal subroutines that are called from Ada.

Calling Ada from foreign languages
7.9

Pragma EXPORT provides a means by which foreign code can make calls to Ada programs. Routines written in Fortran, C, Pascal, and assembler languages can call Ada subprograms, provided that an Ada routine is the main subprogram.

This pragma is allowed to be given only for a library subprogram or for a subprogram declared immediately within a package specification or body that is itself not declared within another subprogram, task, or generic unit. The pragma must be given within the same task or generic unit and in the case of a library subprogram within the same specification or declarative part that contains the subprogram declaration. No more than one EXPORT pragma is allowed for a given subprogram name. If the name denotes more than one subprogram declared earlier within the same package specification or declarative part, a warning is issued and the pragma is ignored.

Parameters used in the exported routine must be of type access or system.address.

The calling foreign language routine must not be multitasked in any way (microtasked, macrotasked, or autotasked), nor should any of its callers be multitasked. The Ada compiler does not necessarily generate reentrant code for subprograms, and reentrance of code is a requirement for multitasking. This restriction applies to only multitasking and not to Cray Ada tasking. Ada tasks can call foreign language routines, which can subsequently call back into Ada, without any problems.

Note

The EXPORT pragma does not allow the specification of language unlike pragma INTERFACE.

The syntax of pragma EXPORT is the following:

```
pragma EXPORT ([NAME =>
<subprogram>
[, [LINK_NAME =>] <string_literal>];
```

`subprogram` This argument must be the simple name of a subprogram, subject to the restrictions specified previously.

The name is allowed to be a name given by a subprogram. Renaming is allowed only if the renaming declaration occurs immediately within the same package specification or declarative as the subprogram that is renamed and an `EXPORT` pragma does not otherwise apply to the subprogram. The name is not allowed to denote a subprogram for which an `INTERFACE` pragma has been specified.

Similarly, an `INTERFACE` pragma may not be specified for a subprogram for which an `EXPORT` pragma has been specified.

`string_literal` This argument is a string literal defining the link name that external languages will use to access the named subprogram. The `link_name` must be specified.

Calling Ada from assembly language 7.9.1

The calling conventions for calling Ada from Cray Assembly Language using pragma `EXPORT` are the same as those for calling Ada from Fortran.

Calling Ada from Fortran 7.9.2

The following example shows a pragma `EXPORT` using Fortran. The main program is Ada, and it calls the Fortran routine, `caller`. `caller` then invokes the Ada subprogram, `exported`. This routine prints out the values. Control then returns to Fortran, then back to Ada, and the program exits.

```
— The program will display the values of
— a, b, c and d, they will be 27, 28, 29 and 30, respectively.
—
— In exported:
— a.all = 27
— b.all = 28
— c.all = 29
— d.all = 30
—
package exporter is
  type aInt is access Integer;
  procedure exported (a, b, c, d : in aInt);
  pragma export (exported, "EXPORTED");
end exported;

with Text_IO;
package body exported is
  procedure exported (a, b, c, d : in aInt) is
    begin
      Text_IO.Put_line("In exported:");
       Text_IO.Put_Line("a.all = "& integer'image(a.all));
       Text_IO.Put_line("b.all = "& integer'image(b.all));
       Text_IO.Put_line("c.all = "& integer'image(c.all));
       Text_IO.Put_line("d.all = "& integer'image(d.all));
  end exported;
end exported;

with System;
with Text_IO;
with exporter;
procedure texport is
  procedure caller (d, e, f, g : in Integer);
  pragma interface ( FORTRAN, caller);
  i, j, k, l : integer;
begin
  i := 27;
  j := 28;
  k := 29;
  l := 30;
  caller (i, j, k, l);
end texport;
```

```
Fortran Source Code:
-------------------

      subroutine caller (I,J,K,L)
      call exported (I,J,K,L)
      end
```

Calling Ada from C 7.9.3

The following is an example of the `EXPORT` pragma using C. The Ada main program calls the C function, `caller`. The C function calls the Ada subprogram `exported`. The `exported` subprogram prints the values. Control is then returned to C, a return value of `0` is sent back to the Ada main program and the value is printed.

```
— The program will display the values of
— a, b, c and d, they will be 27, 28, 29, and 30, respectively.
— Status is equal to zero.
— In exported:
— a = 27
— b = 28
— c = 29
— d = 30
— Status = 0

package exporter is
  type aInt is access Integer;
  procedure exported (a, b, c, d : in aInt);
  pragma export(exported, "exported");
end exporter;

with Text_IO;
package body exporter is
  procedure exported (a, b, c, d : in aInt) is
    begin
      Text_IO.Put_line("In exported:");
       Text_IO.Put_line("a = "& integer'image(a.all)):
       Text_IO.Put_line("b = "& integer'image(b.all));
       Text_IO.Put_line("c = "& integer'image(c.all));
       Text_IO.Put_line("d = "& integer'image(d.all));
  end exported;
end exporter;

with System;
with Text_IO;
with exporter;
procedure texport is
  function caller (d, e, f, g : in integer) return Integer;
  pragma interface (C, caller);
  i, j, k, l, status: integer;
begin
  i := 27;
  j := 28;
  k := 29;
  l := 30;
  status := caller(i, j, k, l);
  Text_IO.Put_line("Status =" & integer'image(status));

end texport;
```

```
C source code:
--------------

short caller(i,j,k,l)
short i,j,k,l;
{
    exported(&i,&j,&k,&l);
    return(0);
}
```

Calling Ada from Pascal
7.9.4

The following is an example of the EXPORT pragma using Pascal. The Ada main program calls the Pascal function, caller. The caller then calls the Ada subprogram, exportpascal. The exportpascal subprogram prints the values. Control is then returned to Pascal. A value of 0 is returned to the Ada main program, and the value is printed.

```
— The program will display the values of
— a, b, c and d, they will be 27, 28, 29, and 30, respectively.
— Status is equal to zero.
— In exported:
— a = 27
— b = 28
— c = 29
— d = 30
— Status = 0

package exporter is
  type aInt is access Integer;
  procedure exportpascal (a, b, c, d : in aInt);
  pragma export(exportpascal, "EXPORTPASCAL");
end exporter;

with Text_IO;
package body exporter is
  procedure exportpascal (a, b, c, d : in aInt) is
    begin
      Text_IO.Put_line("In exported:");
      Text_IO.Put_line(" a = " & integer'image(a.all));
      Text_IO.Put_line(" b = " & integer'image(b.all));
      Text_IO.Put_line(" c = " & integer'image(c.all));
      Text_IO.Put_line(" d = " & integer'image(d.all));
  end exportpascal;
end exporter;

with System;
with Text_IO;
with exporter;
procedure texport is
  function caller(d, e, f, g : in Integer) return Integer;
  pragma interface (PASCAL, caller);
  i, j, k, l, status: integer;
begin
  i := 27;
  j := 28;
  k := 29;
  l := 30;
  status := caller(i, j, k, l);
  Text_IO.Put_line(" Status = " & integer'image(status));

end texport;
```

(continued)

```
Pascal Source Code:
------------------

Module mod004;       (*Necessary for creating a.o without a main program*)

procedure exportpascal(a : INTEGER;     (*External Ada routine definition*)
                      b : INTEGER;
                      c : INTEGER;
                      d : INTEGER); EXTERNAL;

function caller(a : INTEGER;
                b : INTEGER;
                c : INTEGER;
                d : (INTEGER) INTEGER; EXPORTED;

begin
    exportpascal(a,b,c,d);
    caller := 0;
end.
```

LRM Annotations [A]

Page

Page

Tables

LRM Annotations [A]

The Cray Ada compiler supports the full ANSI Ada language as defined by the *Reference Manual for the Ada Programming Language* (LRM) publication ANSI/MIL–STD–1815A. This subsection describes the sections of the language that are designated by the LRM as implementation dependent for the compiler and runtime environment. These language-related issues are presented in the order in which they appear in the LRM. Each section answers the corresponding section of questions presented in *Ada-Europe Guidelines for Ada Compiler Specification and Selection* (J. Nissen and B. Wichmann, MPL Report DITC 10/82).

LRM section 2: Lexical elements
A.1

This subsection describes the parts of section 2 of the LRM that are applicable to the Cray Ada implementation.

LRM 2.1: Character set
A.1.1

The host and target character set is the ASCII character set.

LRM 2.2: Lexical elements, separators, and delimiters
A.1.2

The maximum number of characters on an Ada source line is 200.

LRM 2.8: Pragmas
A.1.3

The Cray Ada compiler implements all language-defined pragmas. Pragmas PAGE and LIST are supported in the context of source and error listings.

The implementation-defined pragmas for the Cray Ada compiler and their documentation are as follows:

- COMMENT, described in this subsection
- EXPORT, described on page 160

- IMAGES, described on page 171
- INTERFACE_INFORMATION, described on page 138
- PRESERVE_LAYOUT, described on page 81
- SUPPRESS_ALL, described on page 190
- VECTORIZE_LOOP, described on page 65
- NO_SUPPRESS, described on page 190
- LINKNAME, described on page 140

Pragma COMMENT A.1.3.1

Pragma COMMENT embeds a comment into the object code. Its syntax is as follows:

```
pragma COMMENT (string_literal);
```

The *string_literal* represents the characters to be embedded in the object code. Pragma COMMENT may appear at any location in the source code of a compilation unit except the generic formal part of a generic unit. You may enter any number of comments into the object code by using pragma COMMENT.

Pragma INTERFACE_INFORMATION A.1.3.2

Pragma INTERFACE_INFORMATION provides an interface to any routine whose name can be specified by an Ada string literal. It may appear in any declaration section of a unit. This allows access to routines whose identifiers do not conform to Ada identifier rules. Pragma INTERFACE must always appear immediately before pragma INTERFACE_INFORMATION for the associated program. The syntax is as follows:

```
pragma INTERFACE (subprogram_name);
pragma INTERFACE_INFORMATION (subprogram_name,string_literal);
```

Pragma INTERFACE_INFORMATION takes two arguments. The subprogram name argument was previously specified in a pragma INTERFACE. The second is a *string literal* argument that specifies the exact link name to be used by the code generator in emitting calls to the associated subprogram. See "Interfacing to Other Languages," page 133, for more information.

Pragma OPTIMIZE
A.1.3.2.1

Pragma OPTIMIZE is supported with limitations. The -O option must be specified for this option to take effect.

LRM section 3: Declarations and types

A.2

This subsection describes the parts of LRM section 3 applicable to the Cray Ada implementation.

LRM 3.2.1: Object declarations

A.2.1

The Cray Ada compiler produces warning messages about the use of uninitialized variables if the optimizer is run. The compiler does not reject a program merely because it contains such variables.

LRM 3.5.1: Enumeration types

A.2.2

The theoretical maximum number of elements in an enumeration type is $(2^{46}) - 1$. The actual limit is much lower due to hardware address limits, and can be realized only if generation of the image table for the type has been deferred and there are no references anywhere in the program that would cause the image table to be generated. The image table is actually two tables: a string literal consisting of all the literals concatenated together, and an index table consisting of an array of integers of length (*numer_of_literals* + 1)

It is obvious from this that for a very large enumeration type, the length of the image table can become quite large.

The Cray Ada implementation-defined pragma IMAGES controls the creation and allocation of the image table for an enumeration type. This pragma may appear only in a compilation unit. The syntaxes of this pragma are as follows:

```
pragma IMAGES    (enumeration_type, deferred);
                 (enumeration_type, immediate);
```

The default clause is deferred. This saves space in the literal table by not creating an image table for an enumeration type unless the 'image, 'value, or 'width attribute for the type is used. If one of these attributes is used, an image table is generated in the literal pool of the compilation unit in which the attribute is used. If the attributes are used in more than one compilation unit, more than one image table is generated, eliminating the benefits of deferring the table. Using the default clause, deferred, for all enumeration types lets users declare very large enumeration types. Using the immediate clause generates the literal table.

LRM 3.5.4: Integer types
A.2.3

There is one predefined integer type: INTEGER. The attributes of type INTEGER are shown in Table 11. Using explicit integer type definitions rather than predefined integer types result in more portable code. Types Short_Integer and Long_Integer are not implemented.

Table 11. Attributes of predefined type INTEGER

Attribute	Type INTEGER
'First	–35,184,372,088,832 (–2**45)
'Last	35,184,372,088,831 (2**45)–1
'Size	46
'Width	15

LRM 3.5.5: Operations of discrete types
A.2.4

Cray Ada supports implementation-dependent attributes for discrete types as described on the following pages.

Extended_Image
attribute for discrete types
A.2.4.1

The Extended_Image attribute returns the string image associated with its first parameter (either an integer or enumeration type), based on the appropriate type Text_IO definitions found in the LRM.

Extended_Image
attribute for integer *types*
A.2.4.2

The definition for the Extended_Image attribute for integers states that the value of ITEM will be consistent with the LRM integer Text_IO definition for put: an integer literal with no underlines, no exponents, no leading zeros (but a single zero for the zero value), and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters, leading spaces are first output to make up the difference. See LRM subsections 14.3.7:10 and 14.3.7:11. The Extended_Image attribute has the following syntax when used with integer types:

```
T'Extended_Image(Item, Width, Base, Based, Space_If_Positive)
```

For a prefix T that is a discrete type or subtype, this attribute is a function that may have more than one parameter. Named association cannot be used with any of the parameters. Parameter *Item* must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

The parameter descriptions for the Extended_Image function for integer types are as follows:

Parameter	Description
Item	Item for which an image is desired. This parameter is required.
Width	Minimum number of characters to be in the returned string. If a width is not specified, the default is 0.
Base	Base in which the image is to be displayed. If a base is not specified, the default is 10.
Based	Specifies (true or false) whether the returned string is in base notation. If a preference is not specified, the default is false.

Parameter Description

Space_If_Positive

User may specify whether the sign bit of a positive integer is included in the string returned. If a preference is not specified, the default is `false`.

Example of `Extended_Image` *for* integer *types* A.2.4.3

To see how this attribute can be used, suppose the following subtype were declared:

```
Subtype T is Integer Range -10..16;
```

The following would then be true:

```
T'Extended_Image(5)                         = "5"
T'Extended_Image(5,0)                       = "5"
T'Extended_Image(5,2)                       = " 5"
T'Extended_Image(5,0,2)                     = "101"
T'Extended_Image(5,4,2)                     = " 101"
T'Extended_Image(5,0,2,True)                = "2#101#"
T'Extended_Image(5,0,10,False)              = "5"
T'Extended_Image(5,0,10,False,True)         = " 5"
T'Extended_Image(-1,0,10,False,False)       = "-1"
T'Extended_Image(-1,0,10,False,True)        = "-1"
T'Extended_Image(-1,1,10,False,True)        = "-1"
T'Extended_Image(-1,0,2,True,True)          = "-2#1#"
T'Extended_Image(-1,10,2,True,True)         = "-2#1#"
```

`Extended_Image` *attribute for enumeration types* A.2.4.4

The definition for the `Extended_Image` attribute for enumeration types states that, given an enumeration literal, the `Extended_Image` attribute outputs the value of the enumeration literal (either an identifier or a character literal) consistent with the LRM integer `Text_IO` definitions for enumeration types. The character case parameter is ignored for character literals. See LRM subsection 14.3.9:9. The `Extended_Image` attribute has the following syntax when used with `enumeration` types:

```
T'Extended_Image(Item, Width, Uppercase)
```

For a prefix `T` that is a discrete type or subtype, this attribute is a function that may have more than one parameter. The *Item* argument must be an enumeration value. The image of an enumeration value is the corresponding identifier, which may have character case and return string width specified.

The parameter descriptions for the `Extended_Image` function for enumeration types are as follows:

Parameter	Description
Item	Item for which an image is desired. This parameter is required.
Width	Minimum number of characters to be in the returned string. If a width is not specified, the default is `0`. If the width specified is larger than the image, the return string is padded with trailing spaces. If the width specified is smaller than the image, the default is used, and the image is output completely.
Uppercase	Specifies the case of the returned string. The default, `true`, is uppercase, and `false` is lowercase.

Example of `Extended_Image` *for enumeration types* A.2.4.5

To see how this attribute can be used, suppose the following types were declared:

```
    Type X is (red, green, blue, purple);
    Type Y is ('a', 'B', 'c', 'D');
```

Given the preceding type declarations, the following would be true:

```
X'Extended_Image(red)                = "RED"
X'Extended_Image(red, 4)             = "RED "
X'Extended_Image(red,2)              = "RED"
X'Extended_Image(red,0,false)        = "red"
X'Extended_Image(red,10,false)       = "red"
Y'Extended_Image('a')                = "'a'"
Y'Extended_Image('B')                = "'B'"
Y'Extended_Image('a',6)              = "'a'"
Y'Extended_Image('a',0,true)         = "'a'"
```

Extended_Value *attribute for discrete types* A.2.4.6

The Extended_Value attribute returns the integer or enumeration value associated with a string item in a manner consistent with the LRM Text_IO definition for the discrete type. The Extended_Value attribute begins reading from the beginning of the string as described in LRM subsections 14.3.7.14 and 14.3.9:11 for the get procedure. The Extended_Value attribute has the following syntax:

```
X'Extended_Value(Item)
```

For a prefix X that is a discrete type or subtype, attribute Extended_Value is a function with a single parameter. The actual parameter, *Item*, must be of predefined type string. Any leading or trailing spaces in string X are ignored. In the case where an illegal string is passed, a CONSTRAINT_ERROR is raised.

The parameter description for the Extended_Value function for discrete types is as follows:

Parameter	Description
Item	Parameter of predefined type string. The type of returned value is the base type X.

Example of Extended_Value *for integer types* A.2.4.7

To see how this attribute can be used with integers, suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Given the preceding type declaration, the following would be true:

```
X'Extended_Value("5")                          = 5
X'Extended_Value(" 5")                         = 5
X'Extended_Value("2#101#")                     = 5
X'Extended_Value("-1")                         = -1
X'Extended_Value(" -1")                        = -1
```

Example of Extended_Value *for enumeration types* A.2.4.8

To see how this attribute can be used with enumeration types, Suppose the following type were declared:

```
Type X is (red, green, blue, purple);
```

The following would then be true:

```
X'Extended_Value("red")                  = RED
X'Extended_Value(" green")               = GREEN
X'Extended_Value("    Purple")           = PURPLE
X'Extended_Value(" GreEn   ")            = GREEN
```

Extended_Width
attribute for discrete types
A.2.4.9

The Extended_Width attribute returns the width value of type Natural for a parameter, depending on its type.

Extended_Width
attribute for integer types
A.2.4.10

For a prefix X that is a discrete subtype, the Extended_Width attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of type or subtype X. Its syntax, when used with integer types, is as follows:

X'Extended_Width(*Base, Based, Space_If_Positive*)

The parameter descriptions for the Extended_Width function for integer types are as follows:

Parameter	Description
Base	Base in which the image is to be displayed. If a base is not specified, the default is 10.
Based	Specifies (true or false) whether the returned string is in base notation. If a preference is not specified, the default is false.
Space_If_Positive	User may specify whether the sign bit of a positive integer is included in the string returned. If a preference is not specified, the default is false.

Example of
Extended_Width *for integer types*
A.2.4.11

To see how this attribute can be used with integer types, suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Given the preceding type declaration, the following would be true:

```
X'Extended_Width                  = 3   -- "-10"
X'Extended_Width(10)              = 3   -- "-10"
X'Extended_Width(2)               = 5   -- "10000"
X'Extended_Width(10,True)         = 7   -- "-10#10#"
X'Extended_Width(2,True)          = 8   -- "2#10000#"
X'Extended_Width(10,False,True) = 3   -- " 16"
X'Extended_Width(10,True,False) = 7   -- "-10#10#"
X'Extended_Width(10,True,True)  = 7   -- " 10#16#"
X'Extended_Width(2,True,True)   = 9   -- " 2#10000#"
X'Extended_Width(2,False,True)  = 6   -- " 10000"
```

Extended_Width *attribute for enumeration types*
A.2.4.12

For a prefix X that is a discrete type or subtype, the Extended_Width attribute is a function yielding the maximum image length over all values of enumeration type or subtype X. Its syntax, when used with enumeration types, is as follows:

```
FUNCTION Extended_Width RETURN Natural;
```

There are no parameters to this function. It returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

Example of Extended_Width *for enumeration types*
A.2.4.13

To see how this attribute can be used with enumeration types, suppose the following types were declared:

```
Type X is (red, green, blue, purple);
Type Z is (X1, X12, X123, X1234);
```

The following would then be true:

```
X'Extended_Width                  = 6   -- "purple"
Z'Extended_Width                  = 5   -- "X1234"
```

LRM 3.5.7: Floating-point types
A.2.5

Using the formulas defined in subsection 3.5.7 of the LRM, you would expect MAX_DIGITS to equal 15 on a Cray Research system. On a Cray Research system, floating-point accuracy is defined differently in Ada than in Fortran.

Theoretically, the maximum number of digits of accuracy that can be obtained from a floating-point number with a 48-bit mantissa is 14 (for more information, see subsection 3.5.7 of the LRM). However, the rounding errors introduced by the hardware in performing floating-point division using a reciprocal approximation cause slight variations in the accuracy of the low-order bits. This can affect the accuracy of the last decimal digit of the result, violating the Ada rules of the accuracy of floating-point arithmetic. For this reason, the maximum number of decimal digits was defined as 13, allowing for a mantissa of only 44 bits. This eliminates the accuracy problems encountered from the rounding of the low-order bits.

LRM 3.5.8: Operations of floating-point types

A.2.6

The only predefined floating-point type is FLOAT. The attributes are shown in Table 12. Using explicit real type definitions will lead to more portable code. Types `Short_Float` and `Long_Float` are not implemented.

Table 12. Attributes of predefined type FLOAT

Attribute	Type FLOAT
'Machine_Overflows	TRUE
'Machine_Rounds	TRUE
'Machine_Radix	2
'Machine_Mantissa	45
'Machine_Emax	8191
'Machine_Emin	-8192
'Mantissa	45
'Digits	13
'Size	64
'Emax	180
'Safe_Emax	8190
'Epsilon	5.684341886081E-14
'First	-2.726870339049E+2465

Table 12. Attributes of predefined type FLOAT (continued)

Attribute	Type FLOAT
'Last	2.726870339049E+2465
'Safe_Large	2.726870339049E+2465
'Safe_Small	1.833603867555E-2466
'Large	1.532495540866E+54
'Small	3.262652233999E-55

System-defined attributes for floating-point types
A.2.6.1

Cray Ada supports system-defined attributes for floating-point types as described on the following pages.

`Extended_Digits` *attribute for floating-point types*
A.2.6.2

The `Extended_Digits` attribute returns a number of type Natural, showing the number of digits in the mantissa of model numbers of subtype `X` as if they were expressed in the specified base. `Extended_Digits` has the following syntax:

```
T'Extended_Digits(base)
```

The parameter description for the `Extended_Digits` function for floating-point types is as follows:

Parameter	Description
base	Base in which the image is to be displayed. If a base is not specified, the default is `10`.

`Extended_Digits` *example*
A.2.6.3

Suppose the following type were declared:

```
Type X is digits 5 range -10.0 .. 16.0;
```

The following would then be true:

```
X'Extended_Digits                    = 5
```

`Extended_Image` *attribute for floating-point types* A.2.6.4

The `Extended_Image` attribute returns a string image associated with its floating-point parameter that is consistent with the LRM floating-point `Text_IO` definition for `put`. The `Text_IO` definition states that this attribute returns the value of a floating-point parameter as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of the parameter. If `Exp` is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of the parameter, or is 0 if the value of the parameter has no integer part. See LRM subsections 14.3.8:13 and 14.3.8:15. The `Extended_Image` attribute has the following syntax:

`X'Extended_Image` (*Item*, *Fore*, *Aft*, *Exp*, *Base*, *Based*)

For a prefix `X` that is a discrete type or subtype, this attribute is a function that may have more than one parameter. Parameter *Item* must be a real value. The resulting string is without underlines or trailing spaces.

The parameter descriptions for the `Extended_Image` function for floating-point types are as follows:

Parameter **Description**

Item Item for which an image is desired. This parameter is required.

Fore Minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative, and the base with the # symbol if based notation is specified. If the integer part to be output has fewer characters than specified by *Fore*, leading spaces are output first to make up the difference. If *Fore* is not specified, the default is `2`.

Aft Minimum number of decimal digits after the decimal point. If the delta of the type or subtype is greater than 0.1, *Aft* is 1. If *Aft* is not specified, the default is `X'Digits-1`. If based notation is specified, the trailing # symbol is included in *Aft*.

Parameter	Description
Exp	Minimum number of digits in the exponent will consist of a sign and the exponent. Additionally, it may contain leading zeros. If `Exp` is not specified, the default is `3`. If `Exp` is `0`, an exponent is not used.
Base	Base in which the image is to be displayed. If a base is not specified, the default is `10`.
Based	Specifies (`true` or `false`) whether the returned string is in base notation. If a preference is not specified, the default is `false`.

Example of `Extended_Image` *for floating-point types*
A.2.6.5

Suppose the following type were declared:

```
Type X is digits 5 range -10.0 .. 16.0;
```

The following would then be true:

```
X'Extended_Image(5.0)                 = " 5.0000E+00"
X'Extended_Image(5.0,1)               = "5.0000E+00"
X'Extended_Image(-5.0,1)              = "-5.0000E+00"
X'Extended_Image(5.0,2,0)             = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)           = " 5.0"
X'Extended_Image(5.0,2,0,0,2)         = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)    = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)    = "2#1.1#E+02"
```

`Extended_Value` *attribute for floating-point types*
A.2.6.6

The `Extended_Value` attribute returns a value of a floating-point type or subtype associated with its parameter in the same manner as does floating-point `Text_IO`. The `Text_IO` definition skips leading zeros, reads a plus or minus sign (if present), and then reads the string according to the syntax of a real literal. The return value corresponds to the sequence input. See LRM subsections 14.3.8:9 and 14.3.8:10. The syntax of `Extended_Value` is as follows:

```
X'Extended_Value(Item)
```

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter *Item* must be of predefined type `string`. Any leading or trailing spaces in the string are ignored. When an illegal string is passed, a `CONSTRAINT_ERROR` is raised.

The parameter description for the `Extended_Value` function for floating-point types is as follows:

Parameter	Description
Item	Parameter of predefined type `string`. The type of the returned value is base type X.

Example of `Extended_Value` *for floating-point types*
A.2.6.7

Suppose the following type were declared:

```
Type X is digits 5 range -10.0 .. 16.0;
```

The following would then be true:

```
X'Extended_Value("5.0")          = 5.000E+00
X'Extended_Value("0.5E1")        = 5.000E+00
X'Extended_Value("2#1.01#E2")    = 5.000E+00
```

Fixed-point types
A.2.6.8

The LRM specifies the model numbers of a fixed-point type. In the following, the end points are not model numbers:

```
type My_Fixed is delta 0.25 range -8.0 .. +8.0;
```

The set of model numbers for this type is {–7.75, –7.5, ..., –0.25. 0.0, +0.25, ..., +7.5, +7.75}. This means that it is legal to raise `Constraint_error` when the value `8.0` is assigned to an object of type `My_Fixed`. The expression `My_Fixed'Last` will return +7.75.

LRM 3.5.10: Operations of fixed-point types A.2.7

Cray Ada supports the system-dependent attributes for fixed-point types that are described in this subsection.

`Extended_Aft` *attribute for fixed-point types* A.2.7.1

The `Extended_Aft` attribute returns the natural number representing the minimum number of characters required to represent, in a specified base, the fractional part of a declared fixed-point type (not including the decimal point). `Extended_Aft` has the following syntax:

```
T'Extended_Aft (Base, Based)
```

The parameter descriptions for the `Extended_Aft` function for fixed-point types are as follows:

Parameter	Description
Base	Base in which the image is to be displayed. If a base is not specified, the default is `10`.
Based	Specifies (`true` or `false`) whether the returned value is in base notation. If a preference is not specified, the default is `false`.

Example of `Extended_Aft` *for fixed-point types* A.2.7.2

Suppose the following type were declared:

```
Type X is delta 0.1 range -10.0 .. 17.1;
```

The following would then be true:

```
X'Extended_Aft                 = 1 -- "1" from 0.1
X'Extended_Aft(2)              = 4 -- "0001" from
2#0.0001#
```

`Extended_Fore` *attribute for fixed-point types* A.2.7.3

The `Extended_Fore` attribute returns the natural number representing the minimum number of characters required to represent, in a specified base, the integer part of a declared fixed-point type (not including the decimal point). `Extended_Fore` has the following syntax when used with fixed-point types:

```
T'Extended_Fore(Base, Based)
```

The parameter descriptions for the `Extended_Fore` function for fixed-point types are as follows:

Parameter	Description
Base	Base in which the subtype is to be displayed. If a base is not specified, the default is `10`.
Based	Specifies (`true` or `false`) whether the returned value is in base notation. If a preference is not specified, the default is `false`.

Example of `Extended_Fore` *for fixed-point types* A.2.7.4

Suppose the following type were declared:

```
Type X is delta 0.1 range -10.0 .. 17.1;
```

The following would then be true:

```
X'Extended_Fore                  = 3   -- "-10"
X'Extended_Fore(2)               = 6   -- " 10001"
```

Extended_Image *attribute for fixed-point types* A.2.7.5

The Extended_Image attribute for fixed-point types returns the string image associated with its fixed-point parameter, based on the same definition as that used by floating-point Text_IO. The attribute returns the value of a fixed-point parameter as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of the parameter. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of the parameter, or is 0 if the value of the parameter has no integer part. See LRM subsections 14.3.8:13 and 14.3.8:15. The Extended_Image attribute has the following syntax when used with fixed-point types:

X'Extended_Image(*Item*, *Fore*, *Aft*, *Exp*, *Base*, *Based*)

For a prefix X that is a discrete type or subtype, this attribute is a function that may have more than one parameter. The parameter *Item* must be a real value. The resulting string is without underlines or trailing spaces.

The parameter descriptions for the Extended_Image function for fixed-point types are as follows:

Parameter **Description**

Item Item for which an image is desired. This parameter is required.

Fore Minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative, and the base with the # symbol if based notation is specified. If the integer part to be output has fewer characters than specified by *Fore*, leading spaces are output first to make up the difference. If *Fore* is not specified, the default is 2.

Aft Minimum number of decimal digits after the decimal point. If the delta of the type or subtype is greater than 0.1, *Aft* is 1. If no *Aft* is specified, the default is X'Digits-1. If based notation is specified, the trailing # symbol is included in *Aft*.

Parameter	Description
Exp	Minimum number of digits in the exponent, will consist of a sign and the exponent. Additionally, it may contain leading zeros. If no *Exp* is specified, the default is 0. If *Exp* is 0, no exponent is used.
Base	Base in which the image is to be displayed. If a base is not specified, the default is 10.
Based	Specifies (true or false) whether the returned string is in base notation. If a preference is not specified, the default is false.

Example of Extended_Image *for fixed-point types*
A.2.7.6

Suppose the following type were declared:

```
Type X is delta 0.1 range -10.0 .. 17.0;
```

The following would then be true:

```
X'Extended_Image(5.0)                   = " 5.00"
X'Extended_Image(5.0,1)                 = "5.00"
X'Extended_Image(-5.0,1)                = "-5.00"
X'Extended_Image(5.0,2,0)               = " 5.0"
X'Extended_Image(5.0,2,0,0)             = " 5.0"
X'Extended_Image(5.0,2,0,0,2)           = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)      = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)      = "2#1.1#E+02"
```

Extended_Value *attribute for fixed-point types*
A.2.7.7

The Extended_Value attribute for fixed-point types returns a value of a fixed-point type or subtype associated with its parameter. This parameter is described in the same manner as the fixed-point Text_IO definition; the attribute skips any leading zeros, reads a plus or minus sign (if present), and then reads the string according to the syntax of a real literal. The return value corresponds to the sequence input. See LRM subsections 14.3.8:9 and 14.3.8:10. Extended_Value has the following syntax when used with fixed-point types:

```
X'Extended_Value(Image)
```

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual *Item* argument must be of predefined type string. Any leading or trailing spaces in string X are ignored. When an illegal string is passed, a CONSTRAINT_ERROR is raised.

The parameter description for the Extended_Value function for fixed-point types is as follows:

Parameter	Description
Image	A parameter of predefined type string. The type of the returned value is the base type of the input string.

Example of Extended_Value *for fixed-point types*
A.2.7.8

Suppose the following type were declared:

```
Type X is delta 0.1 range -10.0 .. 17.0;
```

The following would then be true:

```
X'Extended_Value("5.0")                    = 5.0
X'Extended_Value("0.5E1")                  = 5.0
X'Extended_Value("2#1.01#E2")              = 5.0
```

LRM section 4: Names and expressions
A.3

This subsection describes the parts of LRM section 4 applicable to the Cray Ada implementation.

LRM 4.8: Allocators
A.3.1

The action of pragma CONTROLLED is the default action of the Cray Ada compiler. By default, no automatic storage reclamation is done.

LRM 4.10: Universal expressions
A.3.2

There is no limit on the range of literal values for the compiler.

There is no limit on the accuracy of real literal expressions. Real literal expressions are computed by an arbitrary-precision arithmetic package.

LRM section 9: Tasks
A.4

This subsection describes the parts of LRM section 9 that are applicable to the Cray Ada implementation.

LRM 9.6: Delay statements, duration, and time
A.4.1

This implementation uses 46-bit fixed-point numbers to represent type DURATION. The attributes of the DURATION are shown in Table 13.

Table 13. Attributes of type DURATION

Attribute	Value
'Delta	$6.103515625000 \times 10^{-5}$ (2^{-14})
'Small	$6.103515625000 \times 10^{-5}$ (2^{-14})
'First	-86400
'Last	86400
'Size	32
'Safe_large	$1.310719999390 \times 10^{5}$
'Large	$1.310719999390 \times 10^{5}$

LRM 9.8: Priorities
A.4.2

Sixty-four levels of priority are available to associate with tasks through pragma PRIORITY. Predefined subtype Priority is specified in package SYSTEM as follows:

```
    subtype Priority is Integer range 0..63;
```

Currently, the priority assigned to tasks without a pragma PRIORITY specification is 31; that is the following:

```
(System.Priority'First + System.Priority'Last) / 2
```

LRM 9.11: Shared variables
A.4.3

The only restrictions on shared variables are those specified in the LRM. Distributed tasking model does not exist in Cray Ada, so pragma SHARED is supported *de facto*. Any reference to a data object is a synchronization point.

LRM section 10: Program structure and compilation issues
A.5

All main programs are assumed to be parameterless procedures or functions that return an integer result type. The integer value returned is stored in the UNICOS and status variable.

LRM section 11: Exceptions
A.6

This subsection describes the parts of LRM section 11 applicable to the Cray Ada implementation.

LRM 11.1: Exception declarations
A.6.1

`Numeric_Error` is raised for integer or floating-point overflow and for divide-by-zero situations. Floating-point underflow yields a result of 0 without raising an exception.

`Program_Error` and `Storage_Error` are raised by those situations specified in LRM subsection 11.1.

LRM 11.7: Suppressing checks
A.6.2

Pragma SUPPRESS_ALL suppresses all checks. It can appear anywhere that a SUPPRESS pragma can appear (as defined in the LRM), and its scope is the same as that of pragma SUPPRESS. It is the equivalent to the following call to pragma SUPPRESS:

```
pragma SUPPRESS (access_check, discriminant_check,
division_check, elaboration_check,
index_check, length_check,
overflow_check, range_check,
storage_check);
```

NO_SUPPRESS is a pragma defined by Cray Ada that prevents the suppression of checks within a particular scope. It can be used to override pragma SUPPRESS in an enclosing scope. Pragma NO_SUPPRESS is particularly useful when you have a section of code that relies on predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma NO_SUPPRESS has the same syntax as pragma SUPPRESS and may occur in the same places in source code. The syntax for pragma NO_SUPPRESS is as follows:

```
pragma NO_SUPPRESS (identifier[, [ON=>] [name]);
```

identifier Type of check you want to suppress (for example, access_check).

name Name of the object, type/subtype, task unit, generic unit, or subprogram in which the check should be suppressed.

Pragma SUPPRESS_ALL works the same way as pragma SUPPRESS when mixed with pragma NO_SUPPRESS.

If neither SUPPRESS nor NO_SUPPRESS is present in a program, checks are not suppressed. SUPPRESS may also be controlled using the -i option. See *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

If either SUPPRESS or NO_SUPPRESS is present, the compiler uses the pragma that applies to the specific check to determine whether that check is to be made. If both SUPPRESS and NO_SUPPRESS are present in the same scope, the pragma declared last takes precedence. The presence of pragma SUPPRESS or NO_SUPPRESS in the source code takes precedence over the -i option on the command line.

LRM section 13: Representation clauses and implementation-dependent features
A.7

This subsection describes the parts of LRM section 13 applicable to the Cray Ada implementation.

Cray Ada supports most LRM section 13 facilities. The following subsections document the LRM section 13 facilities that are not implemented or that require explanation. Facilities implemented exactly as described in the LRM are not described.

LRM subsections 13.1 through 13.5 discuss *representation clauses*, which let you specify the way objects are represented. If you do not use representation clauses, the compiler assumes default data representations for each type and stores objects as it chooses.

LRM 13.1: Representation clauses
A.7.1

Representation clauses are not supported for derived types. Records that are packed using pragma PACK adhere to the following conventions:

- The allocated size of an element of an array is always a power of 2 (1, 2, 4, ...). The allocated size for record elements is the minimum number of bits required, except for character types (which are always 8).
- Scalar components of records may not cross word boundaries.
- Components that are array types are allocated on a boundary that is a multiple of the size of an array element. Components that are record types are aligned on word boundaries.

Table 14 shows the objects that can and cannot be packed:

Table 14. Packing of objects

Type	Packable	Packing in composites
Unsigned integer	Yes	Elements must be constrained
Signed integer	Yes	Yes
Fixed point	No	No
Float	No	No
Character	Yes	Yes
String	Yes §	Yes
Boolean	Yes	Yes
Enumeration type	Yes	Yes
Task	No	No
Access type	No	No
Records	Yes	Yes §§
Arrays	Yes	Yes §§

§ Strings are packed by default as 1 character per byte.
§§ Subject to the packing rules of the components.

See "Internal representation of packed types," page 86, for an examples illustrating the use of pragma PACK.

LRM 13.2: Length clauses
A.7.2

The following conventions apply to length clauses in Cray Ada:

- Size specification: `T'size`

  The Cray Ada compiler lets users specify the size of certain Ada objects in a length clause. Integer, character, and enumeration types, or arrays of these types are affected by length clauses. Length clauses are not allowed for floating-point types, fixed-point types, and record types.

- Specification of collection size: `T'storage_size`

  The `storage_size` attribute is supported for collections.

- Specification of storage for a task activation:

```
    T'storage_size
```

  The Cray Ada compiler lets users specify the stack size for a task activation by using the `Storage_Size` attribute in a length clause. The size specified is the initial amount of storage allocated for tasks of that type. Using a length clause applied to a task type does not indicate a ceiling on the amount of space allocated to that task, and if that task needs more space during its execution, it will make calls to the system to increase its stack size.

  Length clauses of the form in the following example (`T` is a task type), specify a task's initial stack size allocated at run time. The use of this clause is encouraged in all tasking applications to control the size of applications and to improve their performance. Without this clause, the compiler may use a large default value (consuming inordinate amounts of memory) or, if the default value is too small, the task may make numerous calls for additional stack space (slowing execution).

  `for T'storage_size use` *expression*

- Specification of `small` for a fixed-point type: `T'small`

  The `small` attribute for fixed-point types is supported only for powers of 2.

LRM 13.3: Enumeration representation clauses
A.7.3

Cray Ada supports enumeration representation clauses for all types other than Boolean.

Be aware that the use of such clauses can introduce considerable overhead into many operations that involve the associated type. Such operations include indexing an array by an element of the type, and computing the `'POS`, `'PRED`, or `'SUCC` attributes for values of the type.

LRM 13.4: Record representation clauses
A.7.4

Record representation clauses are supported by Cray Ada. Because record components are subject to rearrangement by the compiler, you must use representation clauses to provide a particular layout. Such clauses are subject to the following constraints:

- Each component of the record must be specified with a component clause.
- Each component of the record must be constrained such that its size is less than or equal to the number of bits in the component specification.
- The constraint on a record component must be in a positive range. Negative values cannot be represented in record components.
- The alignment of the record is restricted to mod 1, word aligned; therefore, records may be aligned starting at any word address.
- The order of bits within a word is left to right (from most significant to least significant).
- Scalar components may not cross word boundaries.

The following example shows two different representations for the same data items:

```
-- define component types
type Bits_32_Type is range 0..(2**32)-1;

-- define record type
type Rep_Record is record
     A : boolean;
     B : Bits_32_Type;
     C : Bits_32_Type;
end record;

-- one representation
for Rep_Record use Record
     at mod 1;
--   whole record is word-aligned.
       --   no other value is allowed.
     A at 0 range 0..0  ;
     B at 0 range 1..32 ;
     C at 1 range 0..31 ;
--   field can't cross 64-bit boundary
end record;

-- alternate representation: 32-bit-aligned
fields
for Rep_Record use record
     A at 0 range 0..31 ;
     B at 0 range 32..63 ;
     C at 1 range 0..31 ;
end record;
```

Records may be packed by use of pragma PACK. **Packed records follow the convention that scalar components of records do not cross word boundaries.**

LRM 13.5: Address clauses
A.7.5

Address clauses for subprograms, packages, and tasks are not supported, but they are supported for objects and entries.

For address clauses applied to objects, a simple expression of type `address` is interpreted as a position within the linear address space of the machine. The address of an object that is not aligned on a word boundary (for example, a string slice that begins in the third byte of a word) returns the address of the start of the word, rather than the exact address of the object. In general terms, this means that an expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.

Address clauses for objects may be used to access known memory locations. For the Cray Ada Environment, literal addresses are represented as integers, so an unchecked conversion must be applied to these literals before they can be passed as parameters of type `system.address`. The `Location` function is declared in `system` for this purpose.

The following example shows a hypothetical case of an application that defined a specific memory location to hold a status value. An address clause is used to map the `Status_Value` to the predefined location.

```
PACKAGE Application_Status_Manager IS

  -- Set status of application in communication area.
  PROCEDURE Set_Status(Status : IN Natural);

  -- Fetch status of application from communication area.
  FUNCTION Status RETURN Natural;

END Application_Status_Manager;

WITH System;
PACKAGE BODY Application_Status_Manager IS

  Status_Value : Natural;
  FOR Status_Value USE AT System.Location(8#152#);
  -- An applications status flag has been defined to reside at a
  -- particular location.

  PROCEDURE Set_Status(Status : IN Natural) IS
      -- Set_Status sets the status flag for an application
      -- in the predefined status location.

  BEGIN
      Status_Value := Status;

  END Set_Status;

  FUNCTION Status RETURN Natural IS
      -- Status returns the value of the applications status
      -- read directly from the predefined status location.

  BEGIN

      RETURN Status_Value;

  END Status;

END Application_Status_Manager;
```

LRM 13.6: Change of representation
A.7.6

Changes of representation are not supported for types with record representation clauses.

LRM 13.7: The SYSTEM *package*
A.7.7

The pragmas SYSTEM_NAME, STORAGE_UNIT, and MEMORY_SIZE are supported with limitations. These pragmas are allowed only if the argument to them does not change the existing value specified in the system package.

System-dependent named values
A.7.7.1

Two system-dependent named values are available through a package called System_Info. A WITH must be used to access these value. The package contains the following:

```
Package System_Info is

  Compiler_Version_Reference_Number: Constant := 2.0;
  Compiler_Version: Constant String := "2.0";

end System_Info;
```

LRM 13.7.2: Representation attributes
A.7.7.2

The 'Address attribute is not supported for packages. 'Address is also not supported for constants evaluated at compile time, because the values they represent are substituted into code during compilation. 'Address is available, however, in cases in which the value of the constant cannot be determined at compilation time (such as assignment of the constant by use of a function call). 'Address is supported for labels.

LRM 13.7.3: Representation attributes of real types
A.7.7.3

The representation attributes for the predefined floating-point type FLOAT are shown in Table 12, page 179.

LRM 13.8: Machine code insertions
A.7.8

Machine code insertions are not supported.

LRM 13.9: Interface to other languages
A.7.9

Pragma INTERFACE is supported for the following Cray Research system languages:

- Fortran (CF77)
- C (and therefore, UNICOS system calls)
- Pascal
- Cray Assembly Language (CAL)

Additionally, Cray Research system-defined pragma EXPORT exists to allow for calling Ada routines from other Cray Research languages. See "Calling Ada from foreign languages," page 160.

LRM 13.10.1: Unchecked storage deallocation
A.7.9.1

Unchecked storage deallocation frees memory, making it available for other processes or tasks within the scope of the Ada job that initially requested the memory. Because of the memory management scheme currently used by UNICOS, however, the memory freed by unchecked storage deallocation cannot be returned to the operating system to reduce the overall process size of the Ada job. See "Storage management," page 92, for further information on unchecked deallocation.

LRM 13.10.2: Unchecked type conversions
A.7.9.2

Unchecked conversions are allowed between types (or subtypes) T1 and T2 if the following are true:

- They have the same static size.
- They are not unconstrained array or record types.
- They are not private.
- They are not types with discriminants.

The size used in the unchecked conversion is the 'size of the target, which may not be the same as the static size of the target. Also, unchecked conversion of addresses can be very misleading because only a portion of the bits in the word are address bits (24 bits on CRAY X-MP systems and 32 bits on CRAY Y-MP and CRAY-2 systems). The remaining bits in the word should be considered undefined in this context. Therefore, an unchecked conversion can produce erroneous results.

LRM section 14: Input-Output

A.8

This subsection describes the parts of LRM section 14 applicable to the Cray Ada implementation.

LRM section 14.2.1: File management

A.8.1

The form parameter available with the open and create procedure calls in packages sequential_IO, direct_IO, and text_IO has no effect when specified in the procedure call. The default settings for the file type are always used.

LRM section 14.6: Low-level input and output

A.8.2

Cray Ada does not provide the predefined Low_Level_IO package. The capabilities of this package are provided by the I/O redirection feature of UNICOS.

LRM appendix B: Predefined language pragmas

A.8.3

Table 15 lists each of the Cray Ada predefined language pragmas and, if implemented, the location in this manual of additional information.

Table 15. Predefined Cray Ada pragmas

Pragma	Section	Page
CONTROLLED	LRM 4.8 Allocators	188
ELABORATE	Using pragma ELABORATE	9
INLINE	Using pragma INLINE for optimizing	
	Using pragma INLINE	61
INTERFACE	Pragma INTERFACE for Pascal	159
LIST	Using pragmas PAGE and LIST	§
MEMORY_SIZE	LRM 13.7: The package SYSTEM	199
OPTIMIZE	Pragma OPTIMIZE	171
PACK	LRM 13.1: Representation clauses	192
PAGE	Using pragmas PAGE and LIST	§
PRIORITY	LRM 9.8: Priorities	189
SHARED	LRM 9.11: Shared variables	190

§ See *Cray Ada Environment, Volume 1: Reference Manual*, publication SR–3014.

Table 15. Predefined Cray Ada pragmas
(continued)

Pragma	Section	Page
STORAGE_UNIT	LRM 13.7: The package SYSTEM	199
SUPPRESS	LRM 11.7: Suppressing checks	190
SYSTEM_NAME	LRM 13.7: The package SYSTEM	199

LRM appendix F: Implementation-dependent characteristics
A.9

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F.

The following subsections constitute Appendix F for this implementation.

Implementation-defined pragmas
A.9.1

Cray Ada supports the following implementation-defined pragmas listed in Table 16.

Table 16. Implementation-defined pragmas

Attribute	Section title	Page
COMMENT	Pragma COMMENT	170
EXPORT	Calling Ada from a foreign language	160
IMAGES	LRM 3.5.1: Enumeration types	171
INTERFACE_INFORMATION	Pragma INTERFACE_INFORMATION	138
LINKNAME	Pragma LINKNAME	140
NO_SUPPRESS	LRM 11.7: Suppressing checks	190
PRESERVE_LAYOUT	Pragma PRESERVE_LAYOUT	81
SUPPRESS_ALL	LRM 11.7: Suppressing checks	190
VECTORIZE_LOOP	Pragma VECTORIZE_LOOP	65

`T'Size` *attribute when* `T` *is an integer type*
A.9.2

Caution should be used in applying the `T'Size` attribute to integer types. `T'Size` applied to integer types or subtypes returns the minimum number of bits required to hold any possible object of the type or subtype `T`. The minimum value of `T'Size` for integer types or subtypes is 46 or less. It does not accurately represent the size of an object of that type. For integer types and subtypes, the value is always 64 bits unless it is a packed object in which case it may be less.

Implementation-dependent attributes
A.9.3

Cray Ada supports the following implementation-dependent attributes. These are listed in Table 17, along with their types and the location of their documentation.

Table 17. Implementation-dependent attributes

Attribute	Type	LRM subsection	Page
`T'Extended_Aft`	`fixed`	3.5.10	184
`T'Extended_Digits`	`float`	3.5.8	180
`T'Extended_Fore`	`fixed`	3.5.10	185
`T'Extended_Image`	`integer`	3.5.5	172
	`enumeration`	3.5.5	174
	`float`	3.5.8	181
	`fixed`	3.5.10	186
`T'Extended_Value`	`integer`	3.5.5	176
	`enumeration`	3.5.5	176
	`float`	3.5.8	182
	`fixed`	3.5.10	187
`T'Extended_Width`	`integer`	3.5.5	177
	`enumeration`	3.5.5	178

Package `SYSTEM`
A.9.4

The current specification of package `SYSTEM` for CRAY Y-MP systems is as follows:

```
with Unchecked_Conversion;

package System is


  Type NAME is (CRAY_YMP):

  System_Name  : constant name := CRAY_YMP;

  Memory_Size  : constant := (2 ** 32) -1;  --Available memory, in storage units
  Tick         : constant := 0.01;          --Basic clock rate, in seconds




  Storage_Unit : constant := 64;
  Min_Int      : constant := -(2 ** 45);
  Max_Int      : constant := (2 ** 45) -1;
  Max_Digits   : constant := 13;
  Max_Mantissa : constant := 45;
  Fine_Delta   : constant := 1.0 / (2 ** Max_Mantissa);

  subtype Priority is Integer Range 0 .. 63;




  type Memory is private;
  type Address is access Memory;
    --
    -- Ensures compatibility between addresses and access types.
    -- Also provides implicit NULL initial value.
```

(continued)

```
  Null_Address: constant Address := null;
    --
    -- Initial value for any Address object

  type Address_Value is range 0 .. (2**32) -1;
    --
    --A numeric representation of logical addresses for use in address clauses

  function Location is new Unchecked_Conversion (Address_Value, Address);
    --
    -- May be used in address clauses:
    --   Object:  Some_Type;
    --   for Object use at Location (8#4000#);

  function Label ( Name: in String ) return Address;
    --
    -- The LABEL function allows a link name to be specified as the address
    -- for an imported object in an address clause:
    --
    --  Object: Some_Type;
    --  for Object use at Label("Object$$LINK_NAME,EXTERNAL");
    --
    -- System.Label returns Null_Address for non-literal parameters.
    procedure Report_Error

    -- The Report_Error routine can be called from within exception
    -- handlers.  It will print out a traceback for the most recently
    -- handled exception, including a traceback from the point of call
    -- to System.Report_Error itself.

private

.
.
.
.

end System;
```

The SYSTEM package for CRAY X-MP and CRAY-2 systems is the same except for the following four lines:

	CRAY X-MP systems	CRAY-2 systems
type name	CRAY_XMP	CRAY_2
System_Name	CRAY_XMP	CRAY_2
Memory_Size	(2 ** 24) - 1	(2 ** 32) - 1
Address_Value	0 .. (2 ** 24) - 1	0 .. (2 ** 32) - 1

MEMORY_SIZE is a predefined number of words. This number of words is not necessarily the number of words for the machine on which your code is running. To determine the total number of words available or the process size, use UNICOS system call limit(2).

Representation clause restrictions
A.9.5

Restrictions on representation clauses in the Cray Ada compiler are discussed in "LRM section 13: Representation clauses and implementation-dependent features," page 192.

Implementation-generated names
A.9.6

There are no implementation-generated names to denote implementation-dependent components.

Address clause expression interpretation
A.9.7

Expressions that appear in address specifications are interpreted as the first storage unit of the object. "LRM 13.5: Address clauses," page 197, contains a discussion of address clauses.

Unchecked conversion restrictions
A.9.8

Unchecked conversions are allowed between any types or subtypes unless the target type is an unconstrained record or array type. A further discussion of these restrictions on unchecked conversions is available in "LRM 13.10.2: Unchecked type conversions," page 200.

Implementation-dependent characteristics of the I/O packages
A.9.9

The following characteristics of Ada I/O packages are specific to the Cray Ada implementation:

- In Text_IO, type COUNT is defined as follows:

```
type COUNT is range 0..2_147_483_646;
```

- In Text_IO, type Field is defined as follows:

```
subtype Field is integer range 0..1000;
```

- Sequential_IO and Direct_IO cannot be instantiated for unconstrained array types or unconstrained types without default values.

- The standard library contains preinstantiated versions of Text_IO.Integer_IO for type integer and of Text_IO.Float_IO for type float. Additionally, both float and integer instantiations have been provided for CRAY_LIB. It is suggested that the following be used to eliminate multiple instantiations of these packages:

```
Integer_Text_IO
Float_Text_IO
CRAY_MATH_LIB
CRAY_BIT_LIB
CRAY_UTIL_LIB
```

- Multiple files opened to the same external file may be opened only for reading.

- In Text_IO, Direct_IO, or Sequential_IO, calling procedure CREATE with the name of an existing external file does not raise an exception. Instead, it creates a new version of the file.

- In Direct_IO, type COUNT is defined as follows:

```
type COUNT is range 0..35_184_088_831;
```

- According to the latest interpretation of the LRM, during a Text_IO.Get_Line call, if the buffer passed in has been filled, the call is completed and any succeeding characters and/or terminators (such as line, page, or end-of-file markers) are not read. The first Get_Line call reads the line up to, but not including, the end-of-line mark, and the second Get_Line call reads and skips the end-of-line mark left by the first call.